Equation (22) may be proved by inspection of (18), while (21) may be demonstrated by expanding the logarithm in (17) into a Taylor series and retaining only the first two terms. Equations (21) and (22) show that as the number of accesses becomes immaterial, one should use very small resident and overflow records to reduce the total storage volume. Substituting (21) and (22) into eq. (13), we get

$$V^* \sim R \cdot s/a + R \cdot s \cdot (a-1)/a = R \cdot s, \tag{23}$$

which is the absolute minimal storage volume needed, without any "overhead" added by the storage method.

Appendix

PROPERTY 1. For all $a > 1$,

$$p^*(a) < q^*(a). \tag{A1}$$

PROOF. Instead of (A1), we prove the equivalent

$$\exp(q^*/s)/\exp(p^*/s) > 1. \tag{A2}$$

Let us introduce the following notation:

$$b = a + (a^2 - 1)^{\frac{1}{2}}, \quad c = 2(a-1). \tag{A3}$$

Then it is easily seen that

$$q^*/s = 2/(b-1), \tag{A4}$$
$$p^*/s = \ln(b/c). \tag{A5}$$

Substituting (A4) and (A5) into (A2) and expanding the numerator as a Taylor series, retaining the first three terms, we get

$$\frac{\exp(q^*/s)}{\exp(p^*/s)} = \frac{\exp(2/(b-1))}{b/c} > \frac{1 + 2/(b-1) + 2/(b-1)^2}{b/c} = \frac{(b^2+1)c}{(b-1)^2 b} \tag{A6}$$

From definition (A3), we get the following identity:

$$(b-1)^2 = b \cdot c. \tag{A7}$$

Substituting (A7) into (A6), we finally have

$$\frac{\exp(q^*/s)}{\exp(p^*/s)} > \frac{(b^2+1)\cdot c}{b^2 \cdot c} = \frac{b^2+1}{b^2} > 1.$$

Received February 1974; revised March 1975

References

1. Benner, F.H. On designing generalized file records for management information systems. Proc. AFIPS 1967 FJCC, Vol. 31, AFIPS Press, Montvale, N.J., pp. 291–303.
2. *Burroughs DISK FORTE Users Manual.* Burroughs Corp., Detroit, Mich., 1973.
3. Collmeyer, A.J., and Shemer, J.E. Analysis of retrieval performance for selected file organization techniques. Proc. AFIPS 1970 FJCC, Vol. 37, pp. 201–210.
4. Olle, T.W. Generalized systems for storing structured variable length data and retrieving information. In *Mechanized Information Storage, Retrieval and Dissemination*, K. Samuelson, Ed., Rome, 1968.
5. Wilde, D.J., and Beightler, C.S. *Foundations of Optimization.* Prentice-Hall, Englewood Cliffs, N.J., 1967.

Programming Techniques

G. Manacher, S.L. Graham Editors

A Fast Algorithm for Computing Longest Common Subsequences

James W. Hunt
Stanford University
Thomas G. Szymanski
Princeton University

Previously published algorithms for finding the longest common subsequence of two sequences of length n have had a best-case running time of O(n²). An algorithm for this problem is presented which has a running time of O((r + n) log n), where r is the total number of ordered pairs of positions at which the two sequences match. Thus in the worst case the algorithm has a running time of O(n² log n). However, for those applications where most positions of one sequence match relatively few positions in the other sequence, a running time of O(n log n) can be expected.

Key Words and Phrases: Longest common subsequence, efficient algorithms
CR Categories: 3.73, 3.63, 5.25

Copyright © 1977, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

The work of the first author was partially supported by Bell Laboratories' Cooperative Research Fellowship Program. The work of the second author was partially supported by NSF Grants GJ-35570 and DCR74-21939.

Author's addresses: J.W. Hunt, Department of Electrical Engineering, Stanford University, Stanford CA 94305; T.G. Szymanski, Dept. of Electrical Engineering and Computer Science, Princeton University, Brackett Hall, Engineering Quadrangle, Princeton, NJ 98540.

Introduction

Many algorithms [1, 4, 6] for finding the longest common subsequence of two sequences of length n have appeared in the literature. These algorithms all have a worst-case (as well as a best-case) running time of $O(n^2)$.[1]

A more relevant parameter for this problem is r, the total number of matching pairs of positions within the sequences in question. We shall present an $O((r+n)\log n)$ algorithm for the longest common subsequence problem. In the worst case this is of course $O(n^2 \log n)$. However, for a large number of applications, we can expect r to be close to n. In these situations our algorithm will exhibit an $O(n \log n)$ behavior. Typical of such applications are the following:

(1) Finding the longest ascending subsequence of a permutation of the integers from 1 to n [3].
(2) Finding a maximum cardinality linearly ordered subset of some finite collection of vectors in 2-space [7].
(3) Finding the edit distance between two files in which the individual lines of the files are considered to be atomic. The longest common subsequence of these files, considered as sequences, represents that common "core" which does *not* have to be changed if we desire to edit one file into the other.

Thus in the general case our algorithm will not take much longer than the algorithms of [1, 4, 6], whereas in many common applications, our algorithm will perform substantially better.

Let A be a finite sequence of elements chosen from some alphabet. We denote the length of A by $|A|$. $A[i]$ is the ith element of A and $A[i{:}j]$ denotes the sequence $A[i], A[i+1], \cdots, A[j]$.

If U and V are finite sequences, then U is said to be a *subsequence* of V if there exists a monotonically increasing sequence of integers $r_1, r_2, \cdots, r_{|U|}$ such that $U[i] = V[r_i]$ for $1 \le i \le |U|$. U is a *common subsequence* of A and B if U is a subsequence of both A and B. A *longest common subsequence* is a common subsequence of greatest possible length.

Throughout this paper A and B will be used to denote the sequences in question. For ease of presentation, we shall assume both sequences have the same length which will be denoted by n. The number of elements in the set $\{(i, j)$ such that $A[i] = B[j]\}$ will be denoted by r.

Preliminary Results

The key data structure needed by our algorithm is an array of "threshhold values" $T_{i,k}$ defined by $T_{i,k}$ = the smallest j such that $A[1{:}i]$ and $B[1{:}j]$ contain a common subsequence of length k. For example, given sequences $A = abcbdda$, $B = badbabd$ we have $T_{5,1} = 1$, $T_{5,2} = 3$, $T_{5,3} = 6$, $T_{5,4} = 7$, $T_{5,5}$ = undefined.

Each $T_{i,k}$ may thus be considered as a pointer which tells us how much of the B sequence is needed to produce a common subsequence of length k with the first i elements of A.

Note that each row of the T array is strictly increasing; that is,

LEMMA 1. If $T_{i,1}, T_{i,2}, \cdots, T_{i,p}$ *are defined, then* $T_{i,1} < T_{i,2} < \cdots < T_{i,p}$.

PROOF. Consider the common subsequence of length k contained in $A[1{:}i]$ and $B[1{:}T_{i,k}]$. Clearly $B[T_{i,k}]$ is the last member of this common subsequence or else $T_{i,k}$ would not be minimal. Therefore $A[1{:}i]$ and $B[1{:}T_{i,k}-1]$ contain a common subsequence of length $k-1$, that is, $T_{i,k-1} \le T_{i,k} - 1$. □

This linear ordering is of paramount importance in the efficient implementation of our algorithm.

Suppose that we have computed $T_{i,k}$ for all values of k and wish to compute $T_{i+1,k}$ for all values of k. We first show $T_{i+1,k}$ must lie in a specific range of values.

LEMMA 2. $T_{i,k-1} < T_{i+1,k} \le T_{i,k}$.

PROOF. If $A[1{:}i]$ and $B[1{:}T_{i,k}]$ have a common subsequence of length k, then certainly $A[1{:}i+1]$ and $B[1{:}T_{i,k}]$ do also. Thus $T_{i+1,k} \le T_{i,k}$.

By definition, $A[1{:}i+1]$ and $B[1{:}T_{i+1,k}]$ have a common subsequence of length k. Deleting the last element from each of these sequences can remove at most one element from this common subsequence. thus $A[1{:}i]$ and $B[1{:}T_{i+1,k}-1]$ have a common subsequence of length $k-1$. Accordingly $T_{i,k-1} \le T_{i+1,k} - 1$ and $T_{i,k-1} < T_{i+1,k}$. □

The following rule suffices to compute $T_{i+1,k}$ from $T_{i,k-1}$ and $T_{i,k}$.

LEMMA 3.

$$T_{i+1,k} = \begin{cases} \text{smallest } j \text{ such that } A[i+1] = B[i] \\ \qquad \text{and } T_{i,k-1} < j \le T_{i,k} \\ T_{i,k} \text{ if no such } j \text{ exists.} \end{cases}$$

PROOF.

Case 1. No such j exists. By the minimality of $T_{i+1,k}$, any common subsequence of the sequences $A[1{:}i+1]$ and $B[1{:}T_{i+1,k}]$ must have $B[T_{i+1,k}]$ as its last element. Moreover, by Lemma 2 and the premise of this case, $B[T_{i+1,k}]$ does not match $A[i+1]$. Therefor the same common subsequence of length k is also contained in $A[1{:}i]$ and $B[1{:}T_{i+1,k}]$. Thus $T_{i,k} \le T_{i+1,k}$ and by Lemma 2, $T_{i,k}$ must equal $T_{i+1,k}$.

Case 2. There exists a minimal j for which $A[i+1] = B[j]$ and $T_{i,k-1} < j \le T_{i,k}$. Certainly $A[1{:}i+1]$ and $B[1{:}j]$ contain a common subsequence of length k, namely the length $k-1$ common subsequence of

[1] An unpublished result of Michael Paterson shows how to construct an $O(n^2/\log n)$ algorithm for the longest common subsequence problem for sequences over a finite alphabet, and an $O((n^2 \log\log n)/\log n)$ algorithm for sequences over an infinite ordered alphabet. All results of this paper apply to the case of the infinite ordered alphabet.

$A[1{:}i]$ and $B[T_{i,k-1}]$ with the pair $A[i+1]$, $B[j]$ "tacked" onto the end. Thus $T_{i+1,k} \leq j$.

Assume temporarily that $T_{i+1,k} < j$. Since Lemma 2 guarantees that $T_{i,k-1} < T_{i+1,k}$ we can conclude that the last element of the length k common subsequence of $A[1{:}i+1]$ and $B[1{:}T_{i+1,k}]$ does *not* match $A[i+1]$. Thus $A[1{:}i]$ and $B[1{:}T_{i+1,k}]$ also contain a common subsequence of length k which implies that $T_{i,k} \leq T_{i+1,k}$. By Lemma 2 then, $T_{i,k} = T_{i+1,k}$. However, by the above assumption and the premise of this case, $T_{i+1,k} < j \leq T_{i,k}$, implying that $T_{i,k} \neq T_{i+1,k}$. This contradiction leads us to conclude that the original assumption of $T_{i+1,k} < j$ is incorrect and hence we must have $T_{i+1,k} = j$. □

We can now present an $O(n^2 \log n)$ algorithm for determining the length of the longest common subsequence. Subsequent refinements will enable us to not only improve the running time to $O((r+n)\log n)$ but also recover the actual longest common subsequence.

Algorithm 1

```
element array A[1:n], B[1:n];
integer array THRESH[0:n];
integer i, j, k;
THRESH[0] := 0;
for i := 1 step 1 until n do
  THRESH[i] := n + 1;
for i := 1 step 1 until n do
  for j := n step − 1 until 1 do
    if A[i] = B[j] then
      begin
        find k such that THRESH[k−1] < j ≤ THRESH[k];
        THRESH[k] := j;
      end;
print largest k such that THRESH[k] ≠ n + 1;
```

The correctness of the algorithm follows from consideration of the invariant relation "$THRESH[k] = T_{i-1,k}$ for all k" which holds at the start of each iteration on i, and the invariant relation "$THRESH[k] = T_{i,k}$ for all k" which holds at the end of each iteration on i.

Since the $THRESH$ array is monotonically increasing (Lemma 1) we can utilize a binary search to implement the "find" operation in time $O(\log n)$. Thus Algorithm 1 may be implemented to run in $O(n^2 \log n)$ time.

Finally, notice that the direction of the loop on j is crucial. Suppose that for some value of i, $A[i]$ matches several different B elements in a given "threshold" interval, say $B[j_1], \cdots, B[j_m]$ with $THRESH[k-1] = T_{i-1,k-1} < j_1 < \cdots < j_m \leq T_{i-1,k} = THRESH[k]$. From Lemma 3, we see that $T_{i,k} = j_1$ and that $THRESH[k]$ should be updated to this value. Since the inner loop of Algorithm 1 considers values of j in decreasing order, each of the values $j_m, j_{m-1}, \cdots, j_1$ will cause $THRESH[k]$ to take on successively smaller values until it is set equal to the desired value of j_1. If instead the loop on j ran upwards from 1 to n, then not only would $THRESH[k]$ be set to j_1, but $THRESH[k+1]$ would be set to j_2, $THRESH[k+2]$ would be set to j_3 and so forth. Since these latter assignments are unwarranted, we see that the loop on j *must* run downwards.

The Algorithm

A small amount of preprocessing will vastly improve the performance of Algorithm 1. The main source of inefficiency in this algorithm is the inner loop on j in which we repeatedly search for elements of the B sequence which match $A[i]$. Linked list techniques obviate the need for this search.

For each position i we need a list of corresponding j positions such that $A[i] = B[j]$. These lists must be kept in decreasing order in j. All positions of the A sequence which contain the same element may be set up to use the same physical list of matching j's; for the sequences $A = abcbdda$, $B = badbabd$ the desired lists are

$MATCHLIST[1] = \langle 5, 2 \rangle$
$MATCHLIST[2] = \langle 6, 4, 1 \rangle$
$MATCHLIST[3] = \langle\ \rangle$
$MATCHLIST[4] = MATCHLIST[2]$
$MATCHLIST[5] = \langle 7, 3 \rangle$
$MATCHLIST[6] = MATCHLIST[5]$
$MATCHLIST[7] = MATCHLIST[1]$.

We can now display our final algorithm.

Algorithm 2

```
element array A[1:n], B[1:n];
integer array THRESH[0:n];
list array MATCHLIST[1:n];
pointer array LINK[1:n];
pointer PTR;
comment Step 1: build linked lists;
for i := 1 step 1 until n do
  set MATCHLIST[i] := ⟨j1, j2, ··· , jp⟩ such that
    j1 > j2 > ··· > jp and A[i] = B[jq] for 1 ≤ q ≤ p;
comment Step 2: initialize the THRESH array;
THRESH[0] := 0;
for i := 1 step 1 until n do
  THRESH[i] := n + 1;
LINK[0] := null;
comment Step 3: compute successive THRESH values;
for i := 1 step 1 until n do
  for j on MATCHLIST[i] do
    begin
      find k such that THRESH[k−1] < j ≤ THRESH[k];
      if j < THRESH[k] then
        begin
          THRESH[k] := j;
          LINK[k] := newnode (i, j, LINK[k−1]);
        end;
    end;
comment Step 4: recover longest common subsequence in reverse
  order;
k := largest k such that THRESH[k] ≠' n + 1;
PTR := LINK[k];
while PTR ≠ null do
  begin
    print (i, j) pair pointed to by PTR;
    advance PTR;
  end;
```


The subroutine *newnode* invoked in step 3 is a subroutine which creates a list node whose fields contain the values of the arguments to *newnode*. These arguments are, respectively, an index of a position in the A sequence, an index of a position in the B sequence, and a pointer to some other list node. The value returned by *newnode* is a pointer to the list node just created.

THEOREM 1. *Algorithm 2 finds and prints a longest common subsequence of the sequences A and B in time* $O((r+n)\log n)$ *and space* $O(r+n)$.

PROOF. Step 1 can be implemented by sorting each sequence while keeping track of each element's original position. We may then merge the sorted sequences creating the $MATCHLIST$s as we go. This step takes a total of $O(n \log n)$ time and $O(n)$ space.

Step 2 clearly takes $O(n)$ time.

The two outer loops of step 3 should be considered as a single loop over all pairs (i, j) such that $A[i] = B[j]$ taken in order of decreasing j within increasing i. In other words, the outer loops of step 3 induce exactly r executions of the innermost statements of step 3. Since these innermost statements involve one binary search plus a few operations which require constant time, we conclude that the time requirement for step 3 is $O(n + r \log n)$.

In this step we also implement a simple backtracking device that will allow us to recover the longest common subsequence. We record each (i, j) pair which causes an element of the $THRESH$ array to change value. Thus whenever $THRESH[k]$ is defined, $LINK[k]$ points to the head of a list of (i, j) pairs describing a common subsequence of length k. Since at most one list node is created per search, Step 3 will require the allocation of at most $O(r)$ list nodes.

In step 4 we recover the actual longest common subsequence. Clearly this takes at most $O(n)$ time. □

We note that certain input sequences such as $A =$ "*aabaabaab* . . ." and $B =$ "*ababab* . . ." cause Algorithm 2 to use $O(r)$ space even if list nodes are reclaimed whenever they become inaccessible. See [4] for an algorithm which never uses more than $O(n)$ space nor less than $O(n^2)$ time.

A Final Note

The key operations in the implementation of our algorithm are the operations of inserting, deleting, and testing membership of elements in a set where all elements are restricted to the first n integers. Peter van Emde Boas has shown that each such operation can be performed in $O(\log \log n)$ time [2]. His data stucture requires $O(n \log \log n)$ time for initialization. Although the necessary algorithms are quite complex, we can use them to present the following theoretical result.

THEOREM 2. (a) *Algorithm 2 can be implemented to have a running time of* $O(r \log \log n + n \log n)$ *over an infinite alphabet.* (b) *Algorithm 2 can be implemented to have a running time of* $O((n+r)\log \log n)$ *over a fixed finite alphabet.* (c) *The longest ascending subsequence of a permutation of the first n integers may be found in* $O(n \log \log n)$ *time.*

PROOF. The problem of part (c) is, of course, equivalent to finding the longest common subsequence of the given permutation and the sequence $1, 2, \cdots, n$. All three parts of the theorem use basically the same algorithm although the implementation of some of the steps varies slightly. We shall present a common analysis.

In all three cases we require $O(n \log \log n)$ time to initialize van Emde Boas's data structures. Step 1 entails a sorting procedure to set up the $MATCHLIST$s. For the infinite alphabet case, this sort can be done in $O(n \log n)$ time. In the other two cases, we can use a distribution sort to create the $MATCHLIST$s in $O(n)$ time. Step 2 takes $O(n)$ time, step 3 takes $O(n + r \log \log n)$ time and step 4 takes $O(n)$ time. Finally, for the permutation case note that each integer appears exactly once in each sequence and thus we have $r = n$. □

Acknowledgments. The authors are indebted to M. Douglas McIlroy who first suggested this problem to us. Harold Stone suggested a variant of the problem (described and solved in [5]) which led to the development of the present algorithm. Alfred V. Aho and Jeffrey D. Ullman provided us with several enlightening conversations including the particular example given following Theorem 1 which shows that our algorithm can require as much as $O(r)$ space. Peter van Emde Boas made several helpful comments on an early draft of this paper.

Received May 1975; revised January 1976

References

1. Chvatal, V., Klarner, D.A., and Knuth, D.E. Selected combinatorial research problems. STAN-CS-72-292, Dep. Comptr. Sci., Stanford U., Stanford, Calif., June 1972.

2. van Emde Boas, P. Preserving order in a forest in less than logarithmic time. 16th Annual Symp. on Foundations Comptr. Sci., Oct. 1975, pp. 75–84.

3. Fredman, M.L. On computing the length of longest increasing subsequences. *Discrete Mathematics 11*, 1 (Jan. 1975), 29–35.

4. Hirschberg, D.S. A linear space algorithm for computing maximal common subsequences. *Comm. ACM 18*, 6 (June 1975), 341–343.

5. Szymanski, T.G. A special case of the maximal common subsequence problem. TR-170, Dep. Electrical Eng., Princeton U., Princeton, N.J., Jan. 1975.

6. Wagner, R.A. and Fischer, M.J. The string-to-string correction problem. *J. ACM 21*, 1 (Jan. 1975), 168–173.

7. Yao, A.C. and Yao, F.F. On computing the rank function for a set of vectors. UIUCDCS-R-75-699, Dep. Comptr. Sci., U. of Illinois at Urbana-Champaign, Urbana, Ill., Feb. 1975.

Algorithmica (1986) 1: 251–266

Algorithmica
© 1986 Springer-Verlag New York Inc.

An $O(ND)$ Difference Algorithm and Its Variations[1]

Eugene W. Myers[2]

Abstract. The problems of finding a longest common subsequence of two sequences A and B and a shortest edit script for transforming A into B have long been known to be dual problems. In this paper, they are shown to be equivalent to finding a shortest/longest path in an edit graph. Using this perspective, a simple $O(ND)$ time and space algorithm is developed where N is the sum of the lengths of A and B and D is the size of the minimum edit script for A and B. The algorithm performs well when differences are small (sequences are similar) and is consequently fast in typical applications. The algorithm is shown to have $O(N+D^2)$ expected-time performance under a basic stochastic model. A refinement of the algorithm requires only $O(N)$ space, and the use of suffix trees leads to an $O(N \log N + D^2)$ time variation.

Key Words. Longest common subsequence, Shortest edit script, Edit graph, File comparison.

1. Introduction. The problem of determining the differences between two sequences of symbols has been studied extensively [1, 8, 11, 13, 16, 19, 20]. Algorithms for the problem have numerous applications, including spelling correction systems, file comparison tools, and the study of genetic evolution [4, 5, 17, 18]. Formally, the problem statement is to find a longest common subsequence or, equivalently, to find the minimum "script" of symbol deletions and insertions that transform one sequence into the other. One of the earliest algorithms is by Wagner and Fischer [20] and takes $O(N^2)$ time and space to solve a generalization they call the string-to-string correction problem. A later refinement by Hirschberg [7] delivers a longest common subsequence using only linear space. When algorithms are over arbitrary alphabets, use "equal–unequal" comparisons, and are characterized in terms of the size of their input, it has been shown that $\Omega(N^2)$ time is necessary [1]. A "Four Russians" approach leads to slightly better $O(N^2 \log\log N/\log N)$ and $O(N^2/\log N)$ time algorithms for arbitrary and finite alphabets, respectively [13]. The existence of faster algorithms using other comparison formats is still open. Indeed, for algorithms that use "less than–equal–greater than" comparisons, $\Omega(N \log N)$ time is the best lower bound known [9].

Recent work improves upon the basic $O(N^2)$ time arbitrary alphabet algorithm by being sensitive to other problem size parameters. Let the output parameter L be the length of a longest common subsequence and let the dual parameter $D = 2(N-L)$ be the length of the shortest edit script. (It is assumed throughout this introduction that both strings have the same length N.) The two best output-

[1] This work was supported in part by the National Science Foundation under Grant MCS82-10096.
[2] Department of Computer Science, University of Arizona, Tucson, AZ 85721, USA.

Received June 11, 1985; revised January 17, 1986. Communicated by David Dobkin.

sensitive algorithms are by Hirschberg [8] and take $O(NL+N \lg N)$ and $O(DL \log N)$ time. An algorithm by Hunt and Szymanski [11] takes $O((R+N) \log N)$ time where the parameter R is the total number of ordered pairs of positions at which the two input strings match. Note that all these algorithms are $\Omega(N^2)$ or worse in terms of N alone.

In practical situations, it is usually the parameter D that is small. Programmers wish to know how they have altered a text file. Biologists wish to know how one DNA stand has mutated into another. For these situations, an $O(ND)$ time algorithm is superior to Hirschberg's algorithms because L is $O(N)$ when D is small. Furthermore, the approach of Hunt and Szymanski [11] is predicated on the hypothesis that R is small in practice. While this is frequently true, it must be noted that R has no correlation with either the size of the input or the size of the output and can be $O(N^2)$ is many situations. For example, if 10% of all lines in a file are blank and the file is compared against itself, R is greater than $0.01N^2$. For DNA molecules, the alphabet size is four implying that R is at least $0.25N^2$ when an arbitrary molecule is compared against itself or a very similar molecule.

In this paper an $O(ND)$ time algorithm is presented. Our algorithm is simple and based on an intuitive edit graph formalism. Unlike others it employs the "greedy" design paradigm and exposes the relationship of the longest common subsequence problem to the single-source shortest path problem. Another $O(ND)$ algorithm has been presented elsewhere [16]. However, it uses a different design paradigm and does not share the following features. The algorithm can be refined to use only linear space, and its expected-case time behavior is shown to be $O(N+D^2)$. Moreover, the method admits an $O(N \log N+D^2)$ time worst-case variation. This is asymptotically superior to previous algorithms [8, 16, 20] when D is $o(N)$.

With the exception of the $O(N \log N+D^2)$ worst-case variation, the algorithms presented in this paper are practical. The basic $O(ND)$ algorithm served as the basis for a new implementation of the UNIX *diff* program [15]. The version usually runs two to four times faster than the System 5 implementation based on the Hunt and Szymanski algorithm [10]. However, there are cases when D is large where their algoirthm is superior (e.g., for files that are completely different, $R=0$ and $D=2N$). The linear space refinement is roughly twice as slow as the basic $O(ND)$ algorithm but still competitive because it can perform extremely large compares that are out of the range of other algorithms. For instance, two 1.5 million byte sequences were compared in less than two minutes (on a VAX 785 running 4.2 BSD UNIX) even though the difference was greater than 500.

2. Edit Graphs. Let $A=a_1a_2 \cdots a_N$ and $B=b_1b_2 \cdots b_M$ be sequences of length N and M, respectively. The *edit graph* for A and B has a vertex at each point in the grid (x, y), $x \in [0, N]$ and $y \in [0, M]$. The vertices of the edit graph are connected by horizontal, vertical, and diagonal directed edges to form a directed acyclic graph. *Horizontal edges* connect each vertex to its right neighbor, i.e., $(x-1, y) \rightarrow (x, y)$ for $x \in [1, N]$ and $y \in [0, M]$. *Vertical edges* connect each vertex

to the neighbor below it, i.e., $(x, y-1) \rightarrow (x, y)$ for $x \in [0, N]$ and $y \in [1, M]$. If $a_x = b_y$ then there is a *diagonal edge* connecting vertex $(x-1, y-1)$ to vertex (x, y). The points (x, y) for which $a_x = b_y$ are called *match points.* The total number of match points between A and B is the parameter R characterizing the Hunt and Szymansk algorithm [11]. It is also the number of diagonal edges in the edit graph as diagonal edges are in one-to-one correspondence with match points. Figure 1 depicts the edit graph for the sequences $A = abcabba$ and $B = cbabac$.

A *trace* of length L is a sequence of L match points, $(x_1, y_1)(x_2, y_2) \cdots (x_L, y_L)$, such that $x_i < x_{i+1}$ and $y_i < y_{i+1}$ for successive points (x_i, y_i) and (x_{i+1}, y_{i+1}), $i \in [1, L-1]$. Every trace is in exact correspondence with the diagonal edges of a path in the edit graph from $(0, 0)$ to (N, M). The sequence of match points visited in traversing a path from start to finish is easily verified to be a trace. Note that L is the number of diagonal edges in the corresponding path. To construct a path from a trace, take the sequence of diagonal edges corresponding to the match points of the trace and connect successive diagonals with a series of horizontal and vertical edges. This can always be done as $x_i < x_{i+1}$ and $y_i < y_{i+1}$ for successive match points. Note that several paths differing only in their

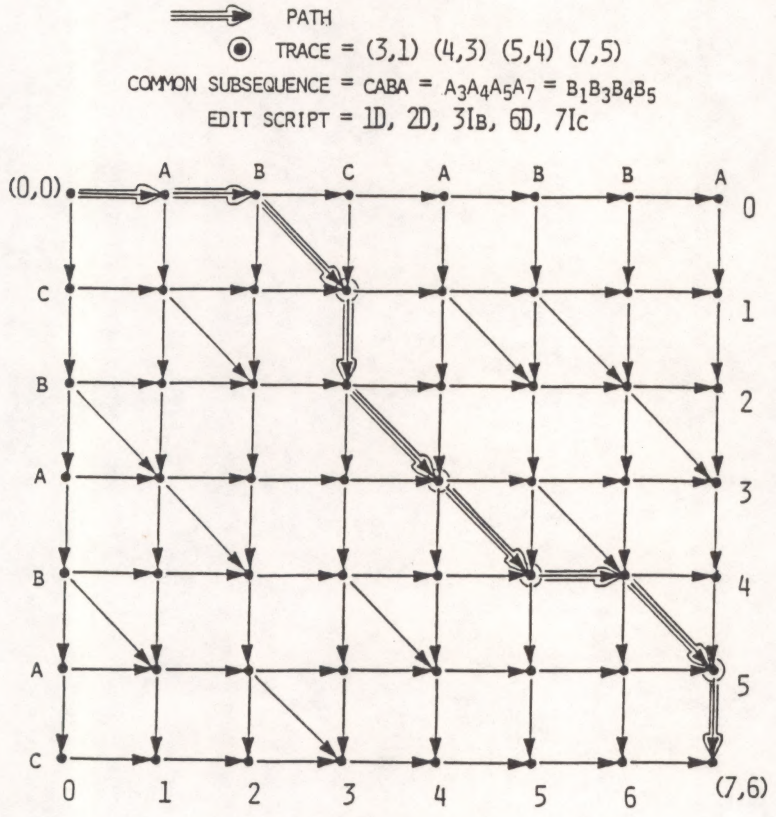


Fig. 1. An edit graph.

nondiagonal edges can correspond to a given trace. Figure 1 illustrates this relation between paths and traces.

A *subsequence* of a string is any string obtained by deleting zero or more symbols from the given string. A *common subsequence* of two strings, A and B, is a subsequence of both. Each trace gives rise to a common subsequence of A and B and vice versa. Specifically, $a_{x_1}a_{x_2}\cdots a_{x_L} = b_{y_1}b_{y_2}\cdots b_{y_L}$ is a common subsequence of A and B if and only if $(x_1, y_1)(x_2, y_2)\cdots(x_L, y_L)$ is a trace of A and B.

An *edit script* for A and B is a set of insertion and deletion commands that transform A into B. The *delete command* "xD" deletes the symbol a_x from A. The *insert command* "$xIb_1, b_2, \cdots b_t$" inserts the sequence of symbols $b_1\cdots b_t$ immediately after a_x. Script commands refer to symbol positions within A before any commands have been performed. One must think of the set of commands in a script as being executed simultaneously. The length of a script is the number of symbols inserted and deleted.

Every trace corresponds uniquely to an edit script. Let $(x_1, y_1)(x_2, y_2)\cdots(x_L, y_L)$ be a trace Let $y_0 = 0$ and $y_{L+1} = M+1$. The associated script consists of the commands: "xD" for $x \notin \{x_1, x_2, \ldots, x_L\}$, and "$x_kIb_{y_k+1}, \ldots, b_{y_{k+1}-1}$" for k such that $y_k + 1 < y_{k+1}$. The script deletes $N-L$ symbols and inserts $M-L$ symbols. So for every trace of length L there is a corresponding script of length $D = N+M-2L$. To map an edit script to a trace, simply perform all delete commands on A, observe that the result is a common subsequence of A and B, and then map the subsequence to its unique trace. Note that inverting the action of the insert commands gives a set of delete commands that map B to the same common subsequence.

Common subsequences, edit scripts, traces, and paths from $(0, 0)$ to (N, M) in the edit graph are all isomorphic formalisms. The edges of every path have the following direct interpretations in terms of the corresponding common subsequence and edit script. Each diagonal edge ending at (x, y) gives a symbol, a_x $(=b_y)$, in the common subsequence; each horizontal edge to point (x, y) corresponds to the delete command "xD"; and a sequence of vertical edges from (x, y) to (x, z) corresponds to the insert command, "$xIb_{y+1}, \cdots, b_z$". Thus the number of vertical and horizontal edges in the path is the length of its corresponding script, the number of diagonal edges is the length of its corresponding subsequence, and the total number of edges is $N+M-L$. Figure 1 illustrates these observations.

The problem of finding a *longest common subsequence* (LCS) is equivalent to finding a path from $(0, 0)$ to (N, M) with the maximum number of diagonal edges. The problem of finding a *shortest edit script* (SES) is equivalent to finding a path from $(0, 0)$ to (N, M) with the minimum number of nondiagonal edges. These are dual problems as a path with the maximum number of diagonal edges has the minimal number of nondiagonal edges $(D+2L = M+N)$. Consider adding a weight or cost to every edge. Give diagonal edges weight 0 and nondiagonal edges weight 1. The LCS/SES problem is equivalent to finding a minimum-cost path from $(0, 0)$ to (N, M) in the weighted edit graph and is thus a special instance of the single-source shortest path problem.

3. An $O((M+N)D)$ Greedy Algorithm. The problem of finding a shortest edit script reduces to finding a path from $(0, 0)$ to (N, M) with the fewest number of horizontal and vertical edges. Let a D-path be a path starting at $(0, 0)$ that has exactly D nondiagonal edges. A 0-path must consist solely of diagonal edges. By a simple induction, it follows that a D-path must consist of a $(D-1)$-path followed by a nondigaonal edge and then a possibly empty sequence of diagonal edges called a *snake.*

Number the diagonals in the grid of edit graph vertices so that diagonal k consists of the points (x, y) for which $x-y=k$. With this definition the diagonals are numbered from $-M$ to N. Note that a vertical (horizontal) edge with start point on diagonal k has end point on diagonal $k-1$ $(k+1)$ and a snake remains on the diagonal in which it starts.

LEMMA 1. *A D-path must end on diagonal $k \in \{-D, -D+2, \ldots D-2, D\}$.*

PROOF. A 0-path consists solely of diagonal edges and starts on diagonal 0. Hence it must end on diagonal 0. Assume inductively that a D-path must end on diagonal k in $\{-D, -D+2, \ldots D-2, D\}$. Every $(D+1)$-path consists of a prefix D-path, ending on say diagonal k, a nondiagonal edge ending on diagonal $k+1$ or $k-1$, and a snake that must also end on diagonal $k+1$ or $k-1$. It then follows that every $(D+1)$-path must end on a diagonal in $\{(-D)\pm 1, (-D+2)\pm 1, \ldots, (D-2)\pm 1, (D)\pm 1\} = \{-D-1, -D+1, \ldots, D-1, D+1\}$. Thus the result holds by induction. □

The lemma implies that D-paths end solely on odd diagonals when D is odd and even diagonals when D is even.

A D-path is *furthest reaching* in diagonal k if and only if it is one of the D-paths ending on diagonal k whose endpoint has the greatest possible row (column) number of all such paths. Informally, of all D-paths ending in diagonal k, it ends furthest from the origin, $(0, 0)$. The following lemma gives an inductive characterization of furthest reaching D-paths and embodies a greedy principle: furthest reaching D-paths are obtained by greedily extending furthest reaching $(D-1)$-paths.

LEMMA 2. *A furthest reaching 0-path ends at (x, x), where x is $\min(z-1 \parallel a_z \neq b_z$ or $z > M$ or $z > N)$. A furthest reaching D-path on diagonal k can without loss of generality be decomposed into a furthest reaching $(D-1)$-path on diagonal $k-1$, followed by a horizontal edge, followed by the longest possible snake or it may be decomposed into a furthest reaching $(D-1)$-path on diagonal $k+1$, followed by a vertical edge, followed by the longest possible snake.*

PROOF. The basis for 0-paths is straightforward. As noted before, a D-path consists of a $(D-1)$-path, a nondiagonal edge, and a snake. If the D-path ends on diagonal k, it follows that the $(D-1)$-path must end on diagonal $k\pm 1$ depending on whether a vertical or horizontal edge precedes the final snake. The final snake must be maximal, as the D-path would not be furthest reaching if

the snake could be extended. Suppose that the $(D-1)$-path is not furthest reaching in its diagonal. But then a further reaching $(D-1)$-path can be connected to the final snake of the D-path with an appropriate nondiagonal move. Thus the D-path can always be decomposed as desired. □

Given the end points of the furthest reaching $(D-1)$-paths in diagonal $k+1$ and $k-1$, say (x', y') and (x'', y''), respectively, Lemma 2 gives a procedure for computing the endpoint of the furthest reaching D-path in diagonal k. Namely, take the further reaching of $(x', y'+1)$ and $(x''+1, y'')$ in diagonal k and then follow diagonal edges until it is no longer possible to do so or until the boundary of the edit graph is reached. Furthermore, by Lemma 1 there are only $D+1$ diagonals in which a D-path can end. This suggests computing the endpoints of D-paths in the relevant $D+1$ diagonals for successively increasing values of D until the furthest reaching path in diagonal $N-M$ reaches (N, M).

For $D \leftarrow 0$ **to** $M+N$ **Do**
  For $k \leftarrow -D$ **to** D **in steps of** 2 **Do**
    Find the endpoint of the furthest reaching D-path in diagonal k.
    If (N, M) is the endpoint **Then**
      The D-path is an optimal solution.
      Stop

The outline above stops when the smallest D is encountered for which there is a furthest reaching D-path to (N, M). This must happen before the outer loop terminates because D must be less than or equal to $M+N$. By construction this path must be minimal with respect to the number of nondiagonal edges within it. Hence it is a solution to the LCS/SES problem.

In presenting the detailed algorithm in Figure 2, a number of simple optimizations are employed. An array, V, contains the endpoints of the furthest reaching D-paths in elements $V[-D]$, $V[-D+2], \ldots, V[D-2]$, $V[D]$. By Lemma 1 this set of elements is disjoint from those where the endpoints of the $(D+1)$-paths will be stored in the next iteration of the outer loop. Thus the array V can simultaneously hold the endpoints of the D-paths while the $(D+1)$-path endpoints are being computed from them. Furthermore, to record an endpoint (x, y) in diagonal k it suffices to retain just x because y is known to be $x-k$. Consequently, V is an array of integers where $V[k]$ contains the row index of the endpoint of a furthest reaching path in diagonal k.

As a practical matter the algorithm searches D-paths where $D \leq \text{MAX}$ and if no such path reaches (N, M) then it reports that any edit script for A and B must be longer than MAX in Line 14. By setting the constant MAX to $M+N$ as in the outline above, the algorithm is guaranteed to find the length of the LCS/SES. Figure 3 illustrates the D-paths searched when the algorithm is applied to the example of Figure 1. Note that a fictitious endpoint, $(0, -1)$, set up in Line 1 of the algorithm is used to find the endpoint of the furthest reaching 0-path. Also note that D-paths extend off the left and lower boundaries of the edit graph

```
    Constant MAX∈[0, M+N]
    Var V: Array [−MAX..MAX] of Integer
1.  V[1]←0
2.  For D←0 to MAX Do
3.      For k←−D to D in steps of 2 Do
4.          If k=−D or k≠D and V[k−1]<V[k+1] Then
5.              x←V[k+1]
6.          Else
7.              x←V[k−1]+1
8.          y←x−k
9.          While x<N and y<M and a_{x+1}=b_{y+1} Do (x, y)←(x+1, y+1)
10.         V[k]←x
11.         If x≥N and y≥M Then
12.             Length of an SES is D
13.             Stop
14. Length of an SES is greater than MAX
```

Fig. 2. The greedy LCS/SES algorithm.

proper as the algorithm progresses. This boundary situation is correctly handled by assuming that there are no diagonal edges in this region.

The greedy algorithm takes at most $O((M+N)D)$ time. Lines 1 and 14 consume $O(1)$ time. The inner *For* loop (Line 3) is repeated at most $(D+1)(D+2)/2$ times because the outer *For* loop (Line 3) is repeated $D+1$ times and during its kth iteration the inner loop is repeated at most k times. All the lines within this inner loop take constant time except for the *While* loop (Line 9). Thus $O(D^2)$ time is spent executing Lines 2–8 and 10–13. The *While* loop is iterated once for each diagonal traversed in the extension of furthest reaching paths. But at most $O((M+N)D)$ diagonals are traversed since all D-paths lie between diagonals $-D$ and D and there are at most $(2D+1)$ min (N, M) points within this band. Thus the algorithm requires a total of $O((M+N)D)$ time. Note that just Line 9, the traversal of snakes, is the limiting step. The rest of the algorithm is $O(D^2)$. Furthermore, the algorithm never takes more than $O((M+N)\text{MAX})$ time in the practical case where the threshold MAX is set to a value much less than $M+N$.

The search of the greedy algorithm traces the optimal D-paths among others. But only the current set of furthest reaching endpoints are retained in V. Consequently, only the length of an SES/LCS can be reported in Line 12. To explicitly generate a solution path, $O(D^2)$ space[3] is used to store a copy of V after each iteration of the outer loop. Let V_d be the copy of V kept after the dth iteration. To list an optimal path from $(0, 0)$ to the point $V_d[k]$, first determine whether it is at the end of a maximal snake following a vertical edge from $V_{d-1}[k+1]$ or a horizontal edge from $V_{d-1}[k-1]$. To be concrete, suppose it is $V_{d-1}[k-1]$. Recursively list an optimal path from $(0, 0)$ to this point and then list the vertical edge and maximal snake to $V_d[k]$. The recursion stops when $d=0$ in which case the snake from $(0, 0)$ to $(V_0[0], V_0[0])$ is listed. So with $O(M+N)$ additional

[3] If only $O(D^2)$ space is to be *allocated*, the algorithm is first run to determine D in $O(N)$ space, then the space is allocated, and finally, the algorithm is run again to determine a solution path.

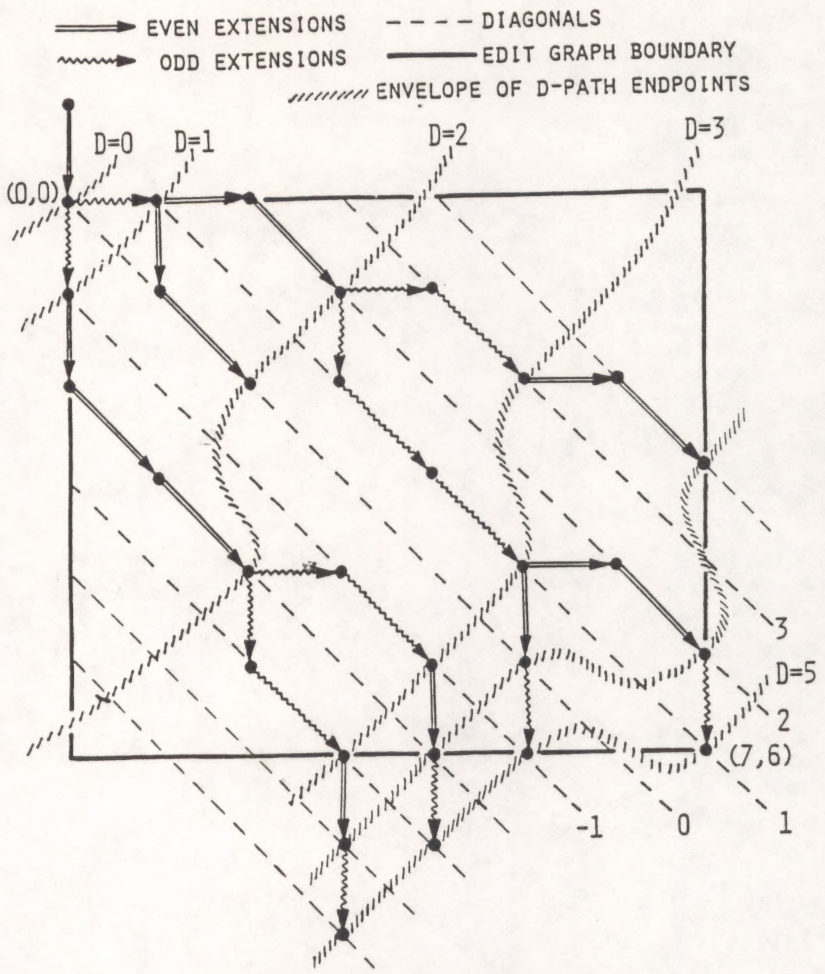


Fig. 3. Furthest reaching paths.

time and $O(D^2)$ space an optimal path can be listed by replacing Line 12 with a call to this recursive procedure with $V_D[N-M]$ as the initial point. A refinement requiring only $O(M+N)$ space is shown in the next section.

As noted in Section 2, the LCS/SES problem can be viewed as an instance of the single-source shortest paths problem on a weighted edit graph. This suggests that an efficient algorithm can be obtained by specializing Dijkstra's algorithm [3]. A basic exercise [2: 207–208] shows that the algorithm takes $O(E \log V)$ time where E is the number of edges and V is the number of vertices in the subject graph. For an edit graph $E < 3V$ since each point has outdegree at most three. Moreover, the $\log V$ term comes from the cost of managing a priority queue. In the case at hand the priorities will be integers in $[0, M+N]$ as edge costs are 0 or 1 and the longest possible path to any point is $M+N$. Under these conditions, the priority queue operations can be implemented in constant time

using "bucketing" and linked-list techniques. Thus Dijkstra's algorithm can be specialized to perform in time linear in the number of vertices in the edit graph, i.e., $O(MN)$. The final refinement stems from noting that all that is needed is the shortest path from the source $(0, 0)$ to the point (M, N). Dijkstra's algorithm determines the minimum distances of vertices from the source in increasing order, one vertex per iteration. By Lemma 1 there are at most $O((M+N)D)$ points less distant from $(0, 0)$ than (M, N) and the previous refinements reduce the cost of each iteration to $O(1)$. Thus the algorithm can stop as soon as the minimum distance to (M, N) is ascertained and it only spends $O((M+N)D)$ time in so doing.

It has been shown that a specialization of Dijkstra's algorithm also gives an $O(ND)$ time algorithm for the LCS/SES problem. However, the resulting algorithm involves a relatively complex discrete priority queue and this queue may contain as many as $O(ND)$ entries even in the case where just the length of the LCS/SES is being computed. While one could argue that further refinement leads to the simple algorithm of this paper, the connection becomes so tenuous that the direct and easily motivated derivation used in this section is preferable. The aim of the discussion is to expose the close relationship between the shortest paths and LCS/SES problems and their algorithms.

4. Refinements. The basic algorithm can be embellished in a number of ways. First, the algorithm's expected performance is $O(M+N+D^2)$, which is much superior to the worst case prediction of $O((M+N)D)$. While not shown here, experiments reveal that the variance about the mean is small especially as the alphabet size becomes large. Thus while there are pathological cases that require $O((M+N)D)$ time they are extremely rare (e.g., like $O(N^2)$ problems for quicksort). Second, the algorithm can be refined to use only linear space when reporting an edit script. The only other algorithm that has been shown to admit such a refinement is the basic $O(MN)$ dynamic programming algorithm [7]. A linear space algorithm is of practical import since many large problems can reasonably be solved in $O(D^2)$ time but not in $O(D^2)$ space. Finally, an $O((M+N)\log(M+N)+D^2)$ worst-case time variation is obtained by speeding up the traversal of snakes with some previously developed techniques [6, 14]. The variation is impractical due to the sophistication of these underlying methods but its superior asymptotic worst-case complexity is of theoretical interest.

4.1. A Probabilisitc Analysis. Consider the following stochastic model for the sequences A and B in a shortest edit script problem. A and B are sequences over an alphabet Σ where each symbol occurs with probability p_σ for $\sigma \in \Sigma$. The N symbols of A are randomly and independently chosen according to the probability densities, p_σ. The $M = N - \delta + \iota$ symbol sequence B is obtained by randomly deleting δ symbols from A and randomly inserting ι randomly chosen symbols. The deletion and insertion positions are chosen with uniform probability. An equivalent model is to generate a random sequence of length $L = N - \delta$ and then randomly insert δ and ι randomly generated symbols to this sequence to

produce A and B, respectively. Note that the LCS of A and B must consist of at least L symbols but may be longer.

An alternate model is to consider A and B as randomly generated sequences of length N and M which are constrained to have an LCS of length L. This model is not equivalent to the one above except in the limit when the size of Σ becomes arbitrarily large and every probability p_σ goes to zero. Nonetheless, the ensuing treatment can also be applied to this model with the same asymptotic results. The first model is chosen as it reflects the edit scripts for mapping A into B that are assumed by the SES problem. While other edit script commands such as "transfers", "moves", and "exchanges" are more reflective of actual editing sessions, their inclusion results in distinct optimization problems from the SES problem discussed here. Hence stochastic models based on such edit process are not considered.

In the edit graph of A and B there are L diagonal edges corresponding to the randomly generated LCS of A and B. Any other diagonal edge, ending at say (x, y), occurs with the same probability that $a_x = b_y$ as these symbols were obtained by independent random trials. Thus the probability of an *off-LCS diagonal* is $\rho = \sum_{\sigma \in \Sigma} p_\sigma^2$. The SES algorithm searches by extending furthest reaching paths until the point (N, M) is reached. Each extension consists of a horizontal or vertical edge followed by the longest possible snake. The maximal snakes consist of a number of LCS and off-LCS diagonals. The probability that there are exactly t off-LCS diagonals in a given extension's snake is $\rho^t(1-\rho)$. Thus the expected number of off-LCS diagonals in an extension is $\sum_{t=0}^{\infty} t\rho^t(1-\rho) = \rho/(1-\rho)$. At most $d+1$ extensions are made in the dth iteration of the outer *For* loop of the SES algorithm. Therefore at most $(D+1)(D+2)\rho/2(1-\rho)$ off-LCS diagonals are traversed in the expected case. Moreover, at most L LCS diagonals are ever traversed. Consequently, the critical *While* loop of the algorithm is executed an average of $O(L+D^2)$ times when ρ is bounded away from 1. The remainder of the algorithm has already been observed to take at worst $O(D^2)$ time. When $\rho = 1$, there is only one letter of nonzero probability in the alphabet Σ, so A and B consists of repetitions of that letter, with probability one. In this case, the algorithm runs in $O(M+N)$ time. Thus the SES algorithm takes $O(M+N+D^2)$ time in the expected case.

4.2. A Linear Space Refinement. The LCS/SES problem is symmetric with respect to the orientation of edit graph edges. Consider reversing the direction of every edge in the edit graph for sequences A and B. Subsequences and edit scripts for A and B are still modeled as paths in this reverse edit graph but now the paths start at (N, M) and end at $(0, 0)$. Also, the interpretation of paths alters just slightly to reflect the reversal of direction. Each diagonal edge *beginning* at (x, y) gives a symbol, a_x $(=b_y)$, in the common subsequence; each horizontal edge *from* point (x, y) corresponds to the delete command "xD"; etc. So the LCS/SES problem can be solved by starting at (N, M) and progressively extending furthest reaching paths in the reverse edit graph until one reaches $(0, 0)$. Hereafter, forward paths will refer to those in the edit graph and reverse paths will refer to those in the reverse edit graph. Since paths in opposing directions

are in exact correspondence, the direction of a path is distinguished only when it is of operational importance.

As in the linear space algorithm of Hirschberg [7], a divide-and-conquer strategy is employed. A D-path has $D+1$ snakes some of which may be empty. The divide step requires finding the $\lceil D/2 \rceil + 1$ or middle snake of an optimal D-path. The idea for doing so is to simultaneously run the basic algorithm in both the forward and reverse directions until furthest reaching forward and reverse paths starting at opposing corners "overlap". Lemma 3 provides the formal observation underlying this approach.

LEMMA 3. *There is a D-path from $(0, 0)$ to (N, M) if and only if there is a $\lceil D/2 \rceil$-path from $(0, 0)$ to some point (x, y) and a $\lfloor D/2 \rfloor$-path from some point (u, v) to (N, M) such that:*

$$\textit{(feasibility)} \quad u+v \geq \lceil D/2 \rceil \quad \text{and} \quad x+y \leq N+M-\lfloor D/2 \rfloor, \quad \textit{and}$$

$$\textit{(overlap)} \quad x-y = u-v \quad \text{and} \quad x \geq u.$$

Moreover, both $D/2$-paths are contained within D-paths from $(0, 0)$ to (N, M).

PROOF. Suppose there is a D-path from $(0, 0)$ to (N, M). It can be partitioned at the start, (x, y), of its middle snake into a $\lceil D/2 \rceil$-path from $(0, 0)$ to (x, y) and a $\lfloor D/2 \rfloor$-path from (u, v) to (N, M) where $(u, v) = (x, y)$. A path from $(0, 0)$ to (u, v) can have at most $u+v$ nondiagonal edges and there is a $\lceil D/2 \rceil$-path to (u, v) implying that $u+v \geq \lceil D/2 \rceil$. A path from (x, y) to (N, M) can have at most $(N+M)-(x+y)$ nondiagonal edges and there is a $\lfloor D/2 \rfloor$-path to (x, y) implying that $x+y \leq N+M-\lfloor D/2 \rfloor$. Finally, $u-v = x-y$ and $u \leq x$ as $(x, y) = (u, v)$.

Conversely, suppose the $\lceil D/2 \rceil$- and $\lfloor D/2 \rfloor$-paths exist. But $u \leq x$ implies there is a k-path from $(0, 0)$ to (u, v) where $k \leq \lceil D/2 \rceil$. By Lemma 1, $\Delta = \lceil D/2 \rceil - k$ is a multiple of 2 as both the k-path and $\lceil D/2 \rceil$-path end in the same diagonal. Moreover, the k-path has $(u+v-k)/2 \geq \Delta/2$ diagonals as $u+v \geq \lceil D/2 \rceil$. By replacing each of $\Delta/2$ of the diagonals in the k-path with a pair of horizontal and vertical edges, a $\lceil D/2 \rceil$-path from $(0, 0)$ to (u, v) is obtained. But then there is a D-path from $(0, 0)$ to (N, M) consisting of this $\lceil D/2 \rceil$-path to (u, v) and the given $\lfloor D/2 \rfloor$-path from (u, v) to (N, M). Note that the $\lfloor D/2 \rfloor$-path is part of this D-path. By a symmetric argument the $\lceil D/2 \rceil$-path is also part of a D-path from $(0, 0)$ to (N, M). □

The outline below gives the procedure for finding the middle snake of an optimal path. For successive values of D, compute the endpoints of the furthest reaching forward D-paths from $(0, 0)$ and then compute the furthest reaching reverse D-paths from (N, M). Do so in V vectors, one for each direction, as in the basic algorithm. As each endpoint is computed, check to see if it overlaps with the path in the same diagonal but opposite direction. A check is needed to ensure that there is an opposing path in the given diagonal because forward paths are in diagonals centered about 0 and reverse paths are in diagonals centered around $\Delta = N - M$. Moreover, by Lemma 1, the optimal edit script length is odd

or even as Δ is odd or even. Thus when Δ is odd, check for overlap only while extending forward paths and when Δ is even, check for overlap only while extending reverse paths. As soon as a pair of opposing and furthest reaching paths overlap, stop and report the overlapping snake as the middle snake of an optimal path. Note that the endpoints of this snake can be readily delivered as the snake was just computed in the previous step.

$\Delta \leftarrow N - M$
For $D \leftarrow 0$ **to** $\lceil (M+N)/2 \rceil$ **Do**
  For $k \leftarrow -D$ **to** D **in steps of** 2 **Do**
    Find the end of the furthest reaching forward D-path in diagonal k.
    If Δ is odd **and** $k \in [\Delta-(D-1), \Delta+(D-1)]$ **Then**
      If the path overlaps the furthest reaching reverse $(D-1)$-path in diagonal k **Then**
        Length of an SES is $2D-1$.
        The last snake of the forward path is the middle snake.
  For $k \leftarrow -D$ to D **in steps of** 2 **Do**
    Find the end of the furthest reaching reverse D-path in diagonal $k+\Delta$.
    If Δ is even **and** $k+\Delta \in [-D, D]$ **Then**
      If the path overlaps the furthest reaching forward D-path in diagonal $k+\Delta$ **Then**
        Length of an SES is $2D$.
        The last snake of the reverse path is the middle snake.

The correctness of this procedure relies heavily on Lemma 3. Without loss of generality suppose Δ is even. The algorithm stops as soon as the smallest D is encountered for which furthest reaching D-paths in opposite directions overlap. First, the overlapping paths must be shown to satisfy the feasibility condition of Lemma 3. Suppose the reverse furthest reaching path ends at (u, v) where $u+v=k$. There is always a k-path of nondiagonal edges from $(0, 0)$ to (u, v) which when combined with the reverse D-path forms a $(k+D)$-path from $(0, 0)$ to (N, M). This path and Lemma 3 imply there are overlapping h-paths where $h = (k+D)/2$ ($k+D$ is divisible by 2 as Δ is even). So certainly there are overlapping furthest reaching h-or-less paths. If $k < D$, then $h < D$ contradicting the fact that furthest reaching D-paths are the first to overlap. So $u+v \geq D$ as desired. A similar argument shows that the furthest reaching forward D-path also satisfies the feasibility constraint of Lemma 3. Now the feasible, overlapping D-paths and Lemma 3 imply that there is a solution path of length $2D$. This must be optimal for if there is a $2k$-path, $k < D$, then the lemma implies there are k-paths that overlap. But this implies that there are overlapping furthest reaching k-or-less paths, contradicting the fact that there are no overlapping paths for smaller D. Thus by Lemma 3, both of the overlapping paths are parts of optimal $2D$-paths from $(0, 0)$ to (N, M). The same conclusion can be drawn when Δ is odd. It remains to show that the chosen snake is the middle snake of the optimal path

of which it is a part. If Δ is odd(even) then this snake is the $(D+1)$st of a forward(reverse) path which is part of an optimal $2D-1(2D)$-path. In either case the snake is the $(\lceil k/2 \rceil + 1)$st of an optimal k-path containing it.

The procedure for finding the middle snake of an optimal D-path requires a total of $O(D)$ working storage for the two V vectors. The procedure only requires $O((M+N)D)$ time because the forward and reverse path extension portions both consume $O((M+N)D)$ time by the same argument used for the basic algorithm. In fact, the number of snakes traversed in extending forward and reverse paths can be *half* that of the basic algorithm because only $D/2$ diagonals are searched. This feature is of practical import—experiments reveal that this procedure for determining the length of an SES is as efficient as the basic algorithm when $D=0$ and rapidly becomes twice as fast as D increases.

Given the middle snake procedure, a linear space algorithm for finding an optimal path through the edit graph of A and B of sizes N and M can be devised. For simplicity, the divide-and-conquer algorithm below just lists a longest common subsequence of A and B. (Producing a shortest edit script is left as an exercise.) First, divide the problem by finding a middle snake from, say, (x, y) to (u, v) of an optimal D-path. Begin conquering the problem, by recursively finding a $\lceil D/2 \rceil$-path from $(0, 0)$ to (x, y) and listing its LCS. Then list the middle snake ("Output $A[x..u]$" lists nothing if $u < x$). Finally, recursively find the $\lfloor D/2 \rfloor$-path from (u, v) to (N, M) and list its LCS. The recursion ends in two ways. If $N=0$ or $M=0$ then $L=0$ and there is nothing to list. In the other case, $N>0$ and $M>0$ and $D \leq 1$. If $D \leq 1$ then B is obtained from A by either deleting or inserting at most one symbol. But then it follows that the shorter of A and B is the LCS and should be listed.

```
LCS(A, N, B, M)
    If N>0 and M>0 Then
        Find the middle snake and length of an optimal path for A
        and B. Suppose it is from (x, y) to (u, v).
        If D>1 Then
            LCS(A1..x], x, B[1..y], y)
            Output A[x+1..u].
            LCS(A[u+1..N], N-u, B[v+1..M], M-v)
        Else If M>N Then
            Output A[1..N].
        Else
            Output B[1..M].
```

Let $T(P, D)$ be the time taken by the algorithm where P is $N+M$. It follows that T satisfies the recurrence inequality:

$$T(P, D) \leq \begin{cases} \alpha PD + T(P_1, \lceil D/2 \rceil) + T(P_2, \lfloor D/2 \rfloor) & \text{if } D>1, \\ \beta P & \text{if } D \leq 1, \end{cases}$$

where $P_1 + P_2 \leq P$ and α and β are suitably large constants. Noting that $\lceil D/2 \rceil \leq 2D/3$ for $D \geq 2$, a straightforward induction argument shows that $T(P, D) \leq 3\alpha PD + \beta P$. Thus the divide-and-conquer algorithm still takes just $O((M+N)D)$ time despite the $\lceil \log D \rceil$ levels of recursion through which it descends. Furthermore, the algorithm only requires $O(D)$ working storage. The middle snake procedure requires two $O(D)$ space V vectors. But this step is completed before engaging in the recursion. Thus only one pair of global V vectors are shared by all invocations of the procedure. Moreover, only $O(\log D)$ levels of recursion are traversed implying that only $O(\log D)$ storage is needed on the recursion stack. Unfortunately, the input sequences A and B must be kept in memory, implying that a total of $O(M+N)$ space is needed.

4.3. An $O((M+N)\log(M+N)+D^2)$ Worst-Case Variation. The final topic involves two previous results, each of which are just sketched here. First, suffix trees [12, 14] are used to efficiently record the common sublists of the sequences being compared. The term sublist is used as opposed to subsequence to emphasize that the symbols must be contiguous. Second, a recent RAM-based algorithm for answering Q on-line queries for the lowest common ancestors of vertices in a fixed V-vertex tree takes $O(V+Q)$ time [6]. The efficient variation centers on quickly finding the length or endpoint of a maximal snake starting at point (x, y). This is shown to reduce to finding the lowest common ancestor of two leaves in a suffix tree. This can be done in $O((M+N)\log(M+N))$ pre-processing time and $O(1)$ time per query using the two techniques above. The ensuing paragraphs embellish these ideas.

A suffix or Patricia tree [12, 14] for a sequence S of length L has edges labelled with sublists of S, has L leaves labelled with the positions of S, and satisfies the following three properties.

1. Concatenating the edge labels traversed on the path from the root to the leaf for position j, gives the suffix, $S[j..L]$, of S starting at j. Thus every path within the tree denotes a suffix of S.
2. Every interior vertex has out-degree greater than one.
3. The labels of the out-edges of every vertex begin with distinct symbols.

These properties can only be satisfied if the last symbol of S is distinct from every other symbol in S. This condition is usually met by appending a special symbol to the target sequence and once satisfied, the suffix tree is unique. Property 2 guarantees that there are less than L interior vertices. Moreover, the substrings labelling edges can be represented by just storing indices to their first and last characters in S. Thus suffix trees can be stored in $O(L)$ space. The efficient construction of suffix trees is beyond the scope of this paper. The reader is referred to a paper by McCreight [14] giving an algorithm that constructs a suffix tree in $O(L)$ steps. Most of the steps are easily done in $O(1)$ time but some require selecting an out-edge based on its first symbol. When the alphabet is finite, the out-degree of vertices is finite and the selection takes $O(1)$ time. When the alphabet is unrestricted, height-balanced trees or some other worst-case efficient search structure permits selection in $O(\log L)$ time. Thus suffix tree construction takes $O(L)$ time for finite alphabets and $O(L \log L)$ time otherwise.

Consider the two paths from the root of S's suffix tree to leaves i and j. Each path from the root to a common ancestor of i and j, denotes a common prefix of the suffixes $S[i..L]$ and $S[j..L]$. From Property 3 it follows that the path to the lowest common ancestor of i and j, denotes the *longest* prefix of their respective suffixes. This observation motivates the following suffix tree characterization of the maximal snake starting at point (x, y) in the edit graph of A and B of lengths N and M, respectively. Form the suffix tree for the sequence $S = A.\$_1.B.\$_2$ where the symbols $\$_1$ and $\$_2$ are not equal to each other or any symbol in A or B. The maximal snake starting at (x, y) is denoted by the path from the root of S's suffix tree to the lowest common ancestor of positions x and $y+N+1$. This follows because neither $\$_1$ or $\$_2$ can be a part of this longest common prefix for the suffixes $A[x..N].\$_1.B.\$_2$ and $B[y..M].\$_2$. So to find the endpoint of a snake starting at (x, y), find the lowest common ancestor of leaves x and $y+N+1$ in the suffix tree and return $(x+m, y+m)$ where m is the length of the sublist denoted by the path to this ancestor. In a linear preprocessing pass the sublist lengths to every vertex are computed and the auxiliary structures needed for the $O(V+Q)$ lowest common ancestor algorithm of Harel and Tarjan [6] are constructed. This RAM-based algorithm requires $O(V)$ preprocessing time but can then answer each on-line query in $O(1)$ time. Thus with $O((M+N)\log(M+N))$ preprocessing time (building the suffix tree is the dominant cost), a collection of on-line queries for the endpoints of maximal snakes can be answered in $O(1)$ time per query.

Modify the basic algorithm of Section 3 by (a) prefacing it with the preprocessing needed for the maximal snake queries and (b) replacing Line 9 with the $O(1)$ query primitives. Recall that every line in the innermost loop other than Line 9 is $O(1)$ and that the loop is repeated $O(D^2)$ times. Now that Line 9 takes $O(1)$ time it follows that this modification results in an algorithm that runs in $O((M+N)\log(M+N)+D^2)$ time. Note that this variation is primarily of theoretical interest. The coefficients of proportionality are much larger for the algorithm fragments employed implying that problems will have to be very large before the variation becomes faster. But suffix trees are particularly space inefficient and two auxiliary trees of equal size are needed for the fast lowest common ancestor algorithm. Thus for problems large enough to make the time savings worthwhile it is likely that there will not be enough memory to accommodate these additional structures.

Acknowledgments. Webb Miller originally proposed the problem of finding an $O(ND)$ algorithm. The author would like to thank him for nurturing this work and his many helpful suggestions. The referees comments and corrections improved the paper greatly.

References

[1] A. V. Aho, D. S. Hirschberg, and J. D. Ullman. Bounds on the complexity of the longest common subsequence problem. *J. ACM*, **23**, 1 (1976), 1–12.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley: Reading, MA, 1983, pp. 203–208.

[3] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.* **1** (1959), 269–271.

[4] J. Gosling. A redisplay algorithm. *Proceedings ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, 1981, pp.

[5] P. A. V. Hall and G. R. Dowling. Approximate string matching. *Comput. Surv.* **12**, 4 (1980), 381–402

[6] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, **13**, 2 (1984), 338–355.

[7] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, **18**, 6 (1975), 341–343.

[8] D. S. Hirschberg. Algorithms for the longest common subsequence problem. *J. ACM* **24**, 4 (1977), 664–675.

[9] D. S. Hirschberg. An information-theoretic lower bound for the longest common subsequence problem. *Inform. Process. Lett.* **7**, 1 (1978), 40–41.

[10] J. W. Hunt and M. D. McIlroy. An algorithm for differential file comparison. Computing Science Technical Report 41, Bell Laboratories (1975).

[11] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, **20**, 5 (1977), 350–353.

[12] D. E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley: Reading, MA, 1983, pp. 490–493.

[13] W. J. Masek and M. S. Paterson. A faster algorithm for computing string edit distances. *J. Comput. System Sci.* **20**, 1 (1980) 18–31.

[14] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM* **23**, 2 (1976), 262–272.

[15] W. Miller and E. W. Myers. A file comparison program. *Software—Practice and Experience*, **15**, 11 (1985), 1025–1040.

[16] N. Nakatsu, Y. Kambayashi and S. Yajima. A longest common subsequence algorithm suitable for similar text strings. *Acta Inform.*, **18** (1982), 171–179.

[17] M. J. Rochkind. The source code control system. *IEEE Trans. Software Engrg.*, **1**, 4 (1975), 364–370.

[18] D. Sankoff and J. B. Kruskal. *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley: Reading, MA, 1983.

[19] W. Tichy. The string-to-string correction problem with block moves. *ACM Trans. Comput. Systems*, **2**, (1984), 309–321.

[20] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM* **21**, 1 (1974), 168–173.

About the ser
The series will p
topical course
Berkeley, and se
of curren math
mathematics wi
mathematics to a
four volumes a
each year.

Volume 1...
Instantons
Daniel S. Freed
This book is bas
M. Freedman
Mathematical S
during its first t
purpose of the
complete and d
Donaldson's proc
certain topologica

Contents
Fake R^4. The Yan
of connections. C
Introduction to
theorem. Compa
The technique
Appendix A: Th
transformatio
Pontrjagin-Thom
Weitzenboeck fo
removability of
Topological remar

1984/232 pp./43 i
ISBN 0-387-9603

Date: Fri, 8 Jul 88 00:46:31 EDT
From: tower@wheaties.ai.mit.edu (Leonard H. Tower Jr.)
To: tower@bu-it.bu.edu
Subject: [tower@bu-it.bu.edu: CS Library resources at Harvard ??]

Return-Path: <tower@bu-it.bu.edu>
Date: Tue, 28 Jun 88 16:49:17 EDT
From: tower@bu-it.bu.edu
To: sudduth@harvard.harvard.edu
Cc: tower@wheaties.ai.mit.edu
Subject: CS Library resources at Harvard ??

GNU is looking for a copy of the report:

> "An O(ND) difference algorithm and its variations",
> E.W.Meyers, Technical Report 85-6, Department of Computer
> Science, University of Arizona, 1985.

Any chance Harvard might have this (MIT and BU don't)? Where (building and library names)? Can a BU employee get in and xerox, or is their a policy against friendly access?

thanx -len

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 15(11), 1025–1040 (NOVEMBER 1985)

A File Comparison Program

WEBB MILLER AND EUGENE W. MYERS
Department of Computer Science, University of Arizona, Tucson, AZ 85721, U.S.A.

SUMMARY

This paper presents a simple method for computing a shortest sequence of insertion and deletion commands that converts one given file to another. The method is particularly efficient when the difference between the two files is small compared to the files' lengths. In experiments performed on typical files, the program often ran four times faster than the UNIX *diff* command.

KEY WORDS Edit distance Edit script File comparison

INTRODUCTION

A file comparison program produces a list of differences between two files. These differences can be couched in terms of lines, e.g. by telling which lines must be inserted, deleted or moved to convert the first file to the second. Alternatively, the list of differences can identify individual bytes. Byte-oriented comparisons are useful with non-text files, such as compiled programs, that are not divided into lines.

The approach adopted here is to generate only instructions to insert or delete entire lines. Since lines are treated as indivisible objects, files can be treated as containing lines consisting of a single symbol. In other words, an n-line file is modelled by a string of n symbols.

In more formal terms, the file comparison problem can be rephrased as follows. The *edit distance* between two strings of symbols is the length of a shortest sequence of insertions and deletions that will convert the first string to the second. The goal, then, is to write a program that computes the edit distance between two arbitrary strings of symbols. In addition, the program must explicitly produce a shortest possible edit script (i.e. sequence of edit commands) for the given strings.

Other approaches have been tried. For example, Tichy[1] discusses a file-comparison tool that determines how one file can be constructed from another by copying blocks of lines and appending lines. However, the ability to economically generate shortest-possible edit scripts depends critically on the repertoire of instructions that are allowed in the scripts.[2]

File comparison algorithms have a number of potential uses beside merely producing a set of edit commands to be read by someone trying to understand the evolution of a program or document. For example, the edit scripts might be text editor instructions that are saved to avoid the expense of storing nearly identical files. Rather than storing

0038–0644/85/111025–16$01.60
© 1985 by John Wiley & Sons, Ltd.
Received 1 October 1984
Revised 21 January 1985

two ... just one of the files and a (presumably short) file containing instructions like

replace lines 6–8 by the line "*s++ = *t++;"

is stored. Rochkind[3] and Tichy[4] discuss version control systems based on this technique.

As further testimony to the range of uses for file comparison techniques, one of the earliest algorithms for the problem was invented simultaneously by biologists interested in comparing long molecules such as proteins and by speech processing experts trying to compare spoken words with known words.[5] These algorithms have also been used in information retrieval systems to cope with spelling mistakes[6] and for video redisplay, where the problem is to send the computer terminal a minimal set of display modification commands that will bring the image up to date.[7]

STRAIGHTFORWARD APPROACHES

The simplest method of file comparison is to look through the two files line by line until they disagree, then search forward in the files until a matching pair of lines is found. Regardless of the strategy for resynchronization, this simple approach suffers from the defect of sometimes producing edit scripts that are much longer than necessary.

To see what goes awry, let the first string consist of n repetitions of the six symbols *axxbxx* and let the second string be derived by adding *bxx* to the front of the first string. Most of the simple resynchronization strategies will match *x*s. For the case $n = 1$, the matching looks as follows:

Simple strategy

```
axxbxx
| | ||
bxxaxxbxx
```

Optimal strategy

```
axxbxx
 \\\\\\
bxxaxxbxx
```

For each of the n segments in the general case, *a* and *b* are deleted, then inserted in the opposite order; *bxx* is inserted at the end. This produces an edit script of length $4n + 3$. On the other hand, the minimal edit script just inserts *bxx* at the front.

In practice, it is common for file comparison programs to require that several consecutive lines match before resynchronization. In the above example, the same edit script of length $4n + 3$ is produced even if resynchronization requires that two contiguous symbols must match. But if three symbols must match before resynchronization is achieved, then the optimal script is found. None the less, the example can be modified to show that any resynchronization strategy that looks for k aligning symbols is not optimal. Moreover, for large k more time is required and matching substrings of length less than k will be missed.

The failure of simple file comparison algorithms is more than just a theoretical curiosity; it can easily happen in practice. For example, suppose that a procedure is added to the beginning of a source file for a program and this file is then compared

against the original file. Where is the first point that lines of the files match? Quite possibly, the match occurs at the ends of the first procedures in each file where there are, for example, several blank lines. If the file comparison program resynchronizes at this point by, in effect, removing the first procedure from each file, then the resulting situation is the same as at the start: *file2* is just *file1* with an additional procedure tacked on the front. Thus the algorithm may report that the two files are entirely different, except for blank lines.

AN EFFICIENT ALGORITHM

The algorithm described below always produces a shortest possible edit script. It works very well on files where the differences are small, and poorly only when the files are quite different. These performance characteristics make for a very efficient comparison tool in the frequently occurring situations where differences are expected to be small. An earlier algorithm[8] enjoys the same general property, but is substantially more complicated. The detailed analysis verifying the correctness and time complexity of the algorithm is deferred until the next section.

All the information needed to compute the edit distance between strings A and B can be determined by comparing every element of A with every element of B. Thus, denoting the length of A by m and the length of B by n, $m \times n$ comparisons suffice. A systematic approach is developed that uses three rules to build up a solution from the solutions to the subproblems that are obtained by considering initial segments of the given strings. Which subproblems need to be solved cannot be determined until the final solution is in hand. This fact is confirmed by the straightforward algorithms, which fail because they fix on specific edit instructions before the two files are completely known.

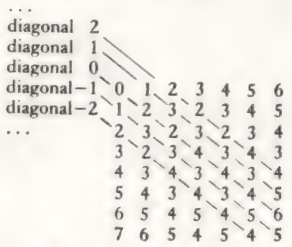
Let $D[i,j]$ be the edit distance between the first i symbols of A (denoted $A[1{:}i]$) and the first j symbols of B (denoted $B[1{:}j]$). $D[i,j]$ makes sense even when i or j is zero; for example $D[i,0]$ is the edit distance between a string of i symbols and a string of 0 symbols, which obviously equals i. These values are arranged as a matrix with $1+m$ rows (one row giving the values $D[0,j]$ and one row for each entry of A) and $1+n$ columns (one column giving the values $D[i,0]$ and one for each entry of B). For example, if $A =$ *abcabba* and $B =$ *cbabac*, the matrix of edit distances is

0	1	2	3	4	5	6	
1	2	3	2	3	4	5	*a*
2	3	2	3	2	3	4	*b*
3	2	3	4	3	4	3	*c*
4	3	4	3	4	3	4	*a*
5	4	3	4	3	4	5	*b*
6	5	4	5	4	5	6	*b*
7	6	5	4	5	4	5	*a*
	c	*b*	*a*	*b*	*a*	*c*	

In this example, the entry $D[5,4]$, which lies at the intersection of row 5 and column

4, (row and column numbers start with 0) is the edit distance between *abcab* (the string labelling rows 1–5) and *cbab* (the string labelling columns 1–4). The value in that position is 3 because *abcab* can be transformed into *cbab* by deleting the leading *a* and the *c* from the first string, then inserting a *c* at the front, but there is no shorter edit script for the transformation. (There may be more than one shortest edit script for a given pair of strings; in this case, delete *a* and *b* and insert a *b*.)

The above matrix exhibits several useful patterns that appear for any choice of the two input strings. The *i*th row (for any *i*) begins with *i*, and each pair of adjacent values differs by 1. The same is true of columns. Diagonals, however, have a different structure. To make this precise, number the diagonals of D as follows.



The values of D along diagonal k begin with $|k|$, then jump to $|k|+2$, then to $|k|+4$, and so on. The validity of this pattern follows from another of particular algorithmic interest: all occurrences of a given value d are on diagonals $-d$, $-d+2$, ..., $d-2$ and d. That is, the entries with value d lie along alternate diagonals in a band of half-width d, centered around diagonal 0. A formal proof of these observations is not given here but can be inferred from the algorithm's proof of correctness given beow.

The file comparison algorithm systematically constructs a solution by using three rules to fill values in the D matrix. Along with each value $D[i,j]$, the algorithm accumulates an edit script of length $D[i,j]$ that converts $A[1:i]$ to $B[1:j]$. The algorithm begins operation by determining all entries in the D matrix that are 0. As is evident, these entries are just the values $D[i,i]$ on diagonal 0 where $A[k] = B[k]$ for all $k \le i$. In other words, the algorithm starts by finding identical prefixes of A and B. Then the algorithm applies the three rules to determine all entries in D that equal 1. Then it fills in the 2s, then the 3s and so on. This continues until the 'south-east' value $D[m,n]$ is determined, at which point the algorithm has found a shortest possible edit script for converting the first input string to the second. In specifying the rules, the notation $A[i]$ denotes the ith symbol of A and $B[j]$ denotes the jth symbol of B.

Rule 1: 'Move right'

Suppose that:

(i) $D[i, j-1]$ (the value just to the left of $D[i, j]$) is known.
(ii) An edit script of length $D[i, j-1]$ that converts $A[1:i]$ to $B[1:j-1]$ is known.
(iii) $D[i, j]$ is unknown.

Then $D[i, j] = D[i, j-1] + 1$, and adding the command 'Insert $B[j]$ after symbol i' to the edit script of (ii) produces a shortest edit script for converting $A[1:i]$ to $B[1:j]$.

If the algorithm has determined the value of $D[i, j-1]$, but not that of $D[i, j]$, then $D[i,j]$ must be greater than $D[i,j-1]$. Consequently, $D[i,j]$ must equal $D[i,j-1]+1$, since the rule shows how to construct a script of this length.

As an example, consider $D[3,3]$, the edit distance from *abc* to *cba* in the sample problem. Suppose that (i) $D[3,2]$, the edit distance from *abc* to *cb*, is 3, that (ii)

Delete symbol 1
Delete symbol 2
Insert *b* after symbol 3

is a shortest possible edit script for converting *abc* to *cb*, and that (iii) $D[3,3]$ has not been filled in. Rule 1 then asserts that $D[3,3]$ must equal 4, and appending the command 'Insert *a* after symbol 3' to the edit script given in (ii) yields a shortest edit script for converting *abc* to *cba*.

Edit commands refer to symbol positions in the *original* string. Thus the second delete command in the script above removes the *b* in *abc* and not the *c*. Also, a sequence of insertions after the same position are assumed to occur in order. Thus the first insert command in the script above places a *b* after the *c* in *abc* and the second 'Insert after symbol 3' places an *a* after the *b* just inserted. These 'parallel' scripts are not sequentially executable by common text editors but are oriented towards users, who cannot retain the intermediate states of a long edit. None the less, these scripts perform the desired transformation if executed sequentially in reverse order.

Rule 2: 'Move down'

Suppose that:

(i) $D[i-1, j]$ (the value just above $D[i, j]$) is known.
(ii) An edit script of length $D[i-1, j]$ that converts $A[1:i-1]$ to $B[1:j]$ is known.
(iii) $D[i, j]$ is unknown.

Then $D[i, j] = D[i-1, j] + 1$, and adding the command 'Delete symbol i' to the edit script of (ii) produces a shortest edit script for converting $A[1:i]$ to $B[1:j]$.

As an example, again consider $D[3,3]$ in the sample problem, which is the edit distance from *abc* to *cba*. Suppose that (i) $D[2,3]$, the edit distance from *ab* to *cba*, is 3, that (ii)

Delete symbol 1
Insert *c* after symbol 1
Insert *a* after symbol 2

is a shortest possible edit script for converting *ab* to *cba*, and that (iii) $D[3,3]$ has not been filled in. Rule 2 then asserts that $D\ [3,3] = 4$, and appending the command 'Delete symbol 3' to the edit script given in (ii) yields a shortest edit script for converting *abc* to *cba*.

Rule 3. 'Slide down the diagonal'

Suppose that

(i) $D[i-1, j-1]$ (the value just above and to the left of $D[i,j]$) is known.

(ii) An edit script of length $D[i-1, j-1]$ that converts $A[1: i-1]$ to $B[1: j-1]$ is known.

(iii) $A[i] = B[j]$.

Then $D[i, j] = D[i-1, j-1]$, and the edit script of (ii) is a shortest edit script for converting $A[1: i]$ to $B[1: j]$.

As an example, look at $D[5,4]$ in the sample problem. Rule 3 says that any edit script of length $D[4,3]$ for *abca* and *cba*, such as

Delete symbol 1
Delete symbol 2
Insert a *b* after symbol 3

is an edit script for *abcab* and *cbab* of length $D[5,4]$

For an illustration of how values of D are determined, suppose that the 0s, 1s and 2s have been filled in the sample matrix of edit distances:

0	1	2	.	.	.	.	
1	2	.	2	.	.	.	*a*
2	.	2	.	2	.	.	*b*
.	2	.	.	.	.	.	*c*
.	.	.	.	.	.	.	*a*
.	.	.	.	.	.	.	*b*
.	.	.	.	.	.	.	*b*
.	.	.	.	.	.	.	*a*
	c	*b*	*a*	*b*	*a*	*c*	

Now apply Rules 1–3 to see how far 3s extend down diagonal 1. Rules 1 and 2 imply that any unfilled position that is immediately to the right of, or immediately below, a 2 must contain a 3. Thus these rules place 3s in positions [1,2], [2,3] and [3,4]. Rule 3 then permits a 3 to fill in the [4,5] location, since the row and column labels there are both *a*, and since this position has a 3 immediately to its north-west. The same sort of reasoning fills in the 3s shown on the next page in diagonals -3, -1 and 3.

The example illustrates that the algorithm finds the d entries of the D matrix by moving right or down from a $d-1$ entry and then making a sequence of diagonal moves. It follows that d values are filled in only on diagonals $-d, -d+2, \ldots, d-2, d$. As already mentioned, all 0s go on diagonal 0. The 1s are filled in by either moving down from diagonal 0 to diagonal -1, or moving right from diagonal 0 to diagonal 1, then sliding down a diagonal.

0	1	2	3	.	.	.	
1	2	3	2	3	.	.	*a*
2	3	2	3	2	3	.	*b*
3	2	3	.	3	.	3	*c*
.	3	.	3	.	3	.	*a*
.	.	3	.	3	.	.	*b*
.	.	.	.	.	.	.	*b*
.	.	.	.	.	.	.	*a*
	c	*b*	*a*	*b*	*a*	*c*	

For the 2s, the algorithm moves right or down from diagonals ± 1, from which it can reach only diagonals -2, 0 and 2, then slides diagonally. Continuing inductively shows the general pattern to be true.

For efficiency, not all the entries of D are explicitly computed and stored. The algorithm only records those d entries nearest the bottom of D in each relevant diagonal (i.e. diagonals $-d$, $-d+2$, ..., $d-2, d$). The notation *last_d*[*k*] indicates the last row containing the most recent value of d to be filled in diagonal k. This simplification is valid since the sought-after corner $D[m,n]$ is such an entry. Furthermore, since the diagonals containing $(d-1)$s alternate with diagonals containing ds, the algorithm can use the same array to hold both the $(d-1)$ entries and the d entries it is computing from them.

Assuming that *last_d*[$k-1$] and *last_d*[$k+1$] indicate the positions of the last $(d-1)$s on diagonals $k-1$ and $k+1$, how is the last d on diagonal k located? The first problem is to find a value of d on diagonal k. One possibiiity is to move right from the last $d-1$ in diagonal $k-1$ to the row *last_d*[$k-1$] on diagonal k. The other possibility is to move down from the last $d-1$ in diagonal $k+1$ to the row *last_d*[$k+1$] + 1 on diagonal k. The algorithm selects the more advantageous of the two moves. Thus if *last_d*[$k+1$] $\geq$ *last_d*[$k-1$] it moves down, otherwise it moves right. (Special care is needed for $k = \pm d$, i.e. for the diagonals that have $(d-1)$s on only one side.) Once on diagonal k, the algorithm slides as far as is permitted by Rule 3.

For an example, return to the problem of filling 3s in diagonal 1 of the sample problem. Just after the 2s have been filled in, *last_d*[0] = 2 and *last_d*[2] = 2. Moving right from diagonal 0 would yield a 3 in row 2 of diagonal 1, whereas moving down from diagonal 2 yields a 3 in row 3. Hence the algorithm applies the move down rule to reach position [3,4], slides down to row 4 using Rule 3, and sets last_d[1] to 4.

Given this method of determining edit distances, producing a corresponding edit script is straightforward. An edit script denoted *script*[k] is associated with each *last_d*[k], and updated according to Rules 1–3. The algorithm fragment that locates the last d on diagonal k is given in Figure 1.

The test

```
if (k = −d or (k ≠ d and last_d[k+1] ≥ last_d[k−1]))
```

```
/* Find a d on diagonal k. */
if (k = -d or (k ≠ d and last_d[k+1] ≥ last_d[k-1])){
    /* Moving down from the last d-1 on diagonal k+1 */
    /* puts you further along diagonal k than does */
    /* moving right from the last d-1 on diagonal k-1. */
    row = last_d[k+1] + 1
    script[k] = script[k+1] with the command
      'Delete the row-th symbol' appended
} else {
    /* Move right from the last d-1 on diagonal k-1. */
    row = last_d[k-1]
    script[k] = script[k-1] with the command
      'Insert B[row+k] after the row-th symbol' appended
}
/* Column where row intersects diagonal k */
col = row + k
/* Slide down the diagonal. */
while (row < m and col < n and A[row+1] =B[col+1]) {
    row = row + 1
    col = col + 1
}
last_d[k] = row
```

Figure 1. Locating the last d on diagonal k

takes special care with the cases $k = \pm d$. When k is $-d$, the algorithm is working on the lowest diagonal that contains ds. The diagonal above it (diagonal $k+1$) contains one or more $(d-1)$s, but the diagonal just below (diagonal $k-1$) contains none, and *last_d*$[k-1]$ is not defined. The clause 'k = −d' guarantees that the move down rule will be applied. Similarly, the clause 'k ≠ d' guarantees that the move right rule is applied when k equals d.

The next step is to enclose the above algorithm fragment in loops that vary d and k appropriately. Perhaps the first pair of looping statements that one might try is:

```
for d = 1,2,3, ...
    for k = -d, -d+2, ..., d-2, d
```

However, an alternative approach (Figure 2) helps to guarantee that array references stay in bounds and further restricts the 'search band' of the algorithm. For example, suppose *row* reaches m, meaning that the algorithm has hit the bottom of the D matrix, say on diagonal k. It is pointless to fill in values to the left of diagonal k, since they cannot contribute to the value of $D[m,n]$. The algorithm arranges that diagonal $k+1$ will be the lowest diagonal that is considered when d is incremented. This is done by using a pair of bounds, *lower* and *upper*, that give the range of diagonals to consider at each iteration. Normally, *lower* is decremented by one and *upper* is incremented by one in each iteration. However, in the situation above the algorithm sets *lower* to $k+2$ and lets


```
/* Initialize: 0 entries in D indicate identical prefixes */
row = min (i:A[i+1] ≠ B [i+1])
last_d[0] = row
script[0] = NULL
if (row = m) lower = 1 else lower = -1
if (row = n) upper = -1 else upper = 1
if (lower > upper)
    Report that the files are identical and terminate execution
/* for each value of the edit distance */
for d = 1,2, ..., max_d {
    /* for each relevant diagonal */
    for k = lower, lower + 2, ..., upper - 2, upper {
        Locate the last d on diagonal k, as in Figure 1.
        if (row = m and col = n)
            Print the edit script pointed to by script[k]
            and terminate execution.
        if (row = m)
            /* Hit last row; don't look to the left. */
            lower = k + 2
        if (col = n)
            /* Hit last column; don't look to the right. */
            upper = k - 2
    }
    lower = lower - 1
    upper = upper + 1
}
Print a message indicating that the edit distance is greater than max_d.
```

Figure 2. The high-level structure of the algorithm

the default decrement set it correctly to $k+1$. The bound *upper* is similarly set when *col* reaches n on a given diagonal.

The algorithm terminates after *max_d* iterations of the outermost loop. If there are no edit scripts of length at most *max_d*, then the algorithm quits and reports this fact. This characteristic is desirable in the common case that one does not want to see the difference if it is too large to be useful. If *max_d* is set to $m + n$, the algorithm finds the difference regardless of its size.

An implementation of the complete algorithm in C is given in the Appendix.

ANALYSIS OF THE ALGORITHM

Correctness

The algorithm employs the simple strategy of finding each d entry by moving right or down from a $d-1$ entry and then making a sequence of diagonal moves. It has yet to be shown that this strategy correctly fills in the D matrix and hence that the algorithm is

correct. An inductive argument is needed to show that at stage d every d entry in the D matrix is found by the strategy. In effect, an argument about *all* edit scripts is required. Certainly, scripts that insert and subsequently delete a given symbol are not optimal and need not be considered. For the remaining scripts, assume without loss of generality that the commands of a script are ordered according to the position in A affected by the command. The scripts produced by Rules 1–3 have this property.

Suppose that S is a shortest edit script of length d for entry $D[i,j]$. For the basis, $d=0$, it follows that $i=j$ and $A[k] = B[k]$ for all $k \leq i$. But the algorithm correctly fills in these entries in its initialization step. Proceeding inductively, suppose that $d > 0$ and let R be the first $d-1$ commands of S. The algorithm's strategy is correct if R is a shortest script for an entry $D[p,q]$ that can reach $D[i,j]$ via a move right or down and a sequence of diagonal moves. Suppose that the last command of S is 'Delete symbol k', where k is between 1 and i. Then the last $i-k$ symbols of $A[1{:}i]$ must match the last $i-k$ symbols of $B[1{:}j]$. Thus S must also be a shortest script for converting $A[1{:}k]$ into $B[1{:}j-(i-k)]$. If it were not, then the shorter script would, by Rule 3, be a shorter script for $D[i,j]$, which is a contradiction. It then follows that R must be a shortest script for converting $A[1{:}k-1]$ into $B[1{:}j-(i-k)]$. Again, if it were not then the shorter script plus the command 'Delete symbol k' would, by Rules 1 and 3, be a shorter script for $D[i,j]$, which is a contradiction. But entry $D[i,j]$ can be reached from $D[k-1, j-(i-k)]$ by a move down and a sequence of diagonal moves. Similar reasoning reveals that if the last command of S is 'Insert an x after symbol k' where k is between 1 and i and x is arbitrary, then R is a shortest script for converting $A[1{:}k]$ to $B[1{:}j-(i-k)-1]$. But again, entry $D[i,j]$ can be reached from $D[k,j-(i-k)-1]$ by a move right and a sequence of diagonal moves. Thus the strategy and algorithm are correct.

A more detailed discussion of the formal issues for the edit distance problem and its generalizations is given in Reference 9.

Efficiency

At worst, the algorithm must determine all the entries of D, hence its running time is at worst proportional to $m \times n$. This is expected, since it is known that for every one of a certain large class of file comparison algorithms there exists some pair of files for which the algorithm takes time proportional to $m \times n$.[10]

The advantage of the algorithm is that it performs efficiently when the size of the output is small compared to m and n. Since the algorithm stops as soon as $D[m,n]$ is filled in, only diagonals $-d$ to d are considered, where $d = D[m,n]$. Thus the running time is proportional to the number of entries on those diagonals, which is less than $(2d+1)\min(m,n)$. Moreover, the expected running time is proportional to $\min(m,n) + d^2$ under appropriate distributional assumptions.[11]

Most file comparison algorithms do not perform this well when the two files are nearly identical. Some take time proportional to $m \times n$ regardless of d.[12] Other algorithms depend heavily on the number r of pairs $[i,j]$ where $A[i]$ equals $B[j]$, which may be large in cases where d is small.[13]

An algorithm whose performance depends on r is used in the UNIX *diff* command.[13, 14] To show how disastrous this dependence on r can be, *diff* was compared to *fcomp*, the new algorithm's implementation as listed in the Appendix. Both programs were run with *file 1* consisting of 1000 blank lines, and *file 2* consisting of *file 1* with a single non-blank line added to both ends. This choice makes $d = 2$ and $r = 10^6$. The *fcomp* program took

about half a second of computer time (on a VAX 11/780), whereas *diff* took over 2·5 minutes.

To turn the tables, the two programs were run on a pair of 1000-line files with no lines in common. This choice makes $d = 2000$ and $r = 0$; *fcomp* ran out of memory after one minute (and reported that the edit distance was at least 617), whereas *diff* solved the problem in about 8 seconds. (This mode of failure for *fcomp* can be avoided since the algorithm can be modified to run in space proportional to $m+n$.[11]) Thus there exist pathological cases where *fcomp*'s performance is much worse than *diff*'s, and vice versa.

To obtain a more realistic idea of how *fcomp* compares with *diff*, figures were gathered for their relative performance comparing 1000-line files of C programs, with various values of d between 5 and 50. Typical values of r were 10,000–20,000. (If one out of every 10 lines were blank, then the pairs of blank lines, one from each file, would contribute 10,000 to r.) On these problems, *fcomp* typically ran about 4 times faster than *diff*. The only circumstances in which this trend was broken were cases when all differences between the files fell in a small range of lines. *diff* begins operation by stripping away lines that match at the fronts and rears of the two files. For example, if all differences between the two files occur in lines 1–100, *diff* quickly reduces the problem to that of comparing two 100-line files, then applies its main algorithm. For such problems, *fcomp* ran about 2–3 times faster than *diff*.

It is notoriously difficult to judge the relative merits of two programs, since performance often depends critically on the data and the programming details. To evaluate the effect of coding differences, *diff*'s underlying algorithm was implemented in the spirit of *fcomp*. In particular, each line of the two files was saved using a call to the storage allocation routine *malloc*. (To conserve space, *diff* stores internally only the lines' hash values. This strategy costs the time to compute those values and to read each file twice, but speeds the process of comparing lines.) Again, *fcomp* was more efficient by a factor of around four. The program listing given in the Appendix will facilitate efforts to corroborate, extend or invalidate this experimental conclusion.

ACKNOWLEDGEMENTS

We would like to thank Chris Fraser, Gary Levin and Titus Purdin for suggesting several improvements to the paper. Walter Tichy pointed out References 5 and 8 and the referees made several helpful suggestions. This work was supported by National Science Foundation under Grants MCS82–10722 and MCS82–10096.

APPENDIX

The following program pp. 1036–1039 implements the file comparison algorithm in C.

```
/*
* fcomp - a file comparison program
*
* A command line has the form
*       fcomp [-n] file1 file2
* where the optional flag  n  is an integer constant that limits the size of
* edit scripts that will be considered by fcomp.  If all edit scripts changing
* file1 to file2 contain more than  n  insertions and deletions, then a message
* to that effect is all that is printed.  If no  n  is specified, then arbitrarily
* long edit scripts are considered.
*/

#include <stdio.h>
#define MAXLINES 2000                   /* maximum number of lines in a file */
#define ORIGIN  MAXLINES                /* subscript for diagonal 0 */
#define INSERT 1
#define DELETE 2

/* edit scripts are stored in linked lists */
struct edit {
        struct edit *link;              /* previous edit command */
        int op;                         /* INSERT or DELETE */
        int line1;                      /* line number in file1 */
        int line2;                      /* line number in file2 */
};

char *A[MAXLINES], *B[MAXLINES];    /* pointers to lines of file1 and file2 */

main(argc, argv)
int argc;
char *argv[];
{
        int     max_d,              /* bound on size of edit script */
                m,                  /* number of lines in file1 */
                n,                  /* number of lines in file2 */
                lower,              /* left-most diagonal under consideration */
                upper,              /* right-most diagonal under consideration */
                d,                  /* current edit distance */
                k,                  /* current diagonal */
                row,                /* row number */
                col;                /* column number */

        /* for each diagonal, two items are saved: */
        int last_d[2*MAXLINES+1];       /* the row containing the last d */
        struct edit *script[2*MAXLINES+1];   /* corresponding edit script */

        struct edit *new;
        char *malloc();

        if (argc > 1 && argv[1][0] == '-') {
                max_d = atoi(&argv[1][1]);
                ++argv;
                --argc;
        } else
                max_d = 2*MAXLINES;
        if (argc != 3)
                fatal("Fcomp requires two file names.");
        /* Read in file1 and file2. */
        m = in_file(argv[1], A);
        n = in_file(argv[2], B);
```

```
/* Initialize: 0 entries in D indicate identical prefixes. */
for (row = 0; row < m && row < n && strcmp(A[row], B[row]) == 0; ++row)
        ;
last_d[ORIGIN] = row;
script[ORIGIN] = NULL;
lower = (row == m) ? ORIGIN + 1 : ORIGIN - 1;
upper = (row == n) ? ORIGIN - 1 : ORIGIN + 1;
if (lower > upper) {
        puts("The files are identical.");
        exit(0);
}

/* for each value of the edit distance */
for (d = 1; d <= max_d ; ++d) {
        /* for each relevant diagonal */
        for (k = lower; k <= upper; k += 2) {
                /* Get space for the next edit instruction. */
                new = (struct edit *) malloc(sizeof(struct edit));
                if (new == NULL)
                        exceed(d);

                /* Find a d on diagonal k. */
                if (k == ORIGIN-d || k != ORIGIN+d && last_d[k+1] >= last_d[k-1]) {
                        /*
                         * Moving down from the last d-1 on diagonal k+1
                         * puts you farther along diagonal k than does
                         * moving right from the last d-1 on diagonal k-1.
                         */
                        row = last_d[k+1]+1;
                        new->link = script[k+1];
                        new->op = DELETE;
                } else {
                        /* Move right from the last d-1 on diagonal k-1. */
                        row = last_d[k-1];
                        new->link = script[k-1];
                        new->op = INSERT;
                }
                /* Code common to the two cases. */
                new->line1 = row;
                new->line2 = col = row + k - ORIGIN;
                script[k] = new;

                /* Slide down the diagonal. */
                while (row < m && col < n && strcmp(A[row], B[col]) == 0) {
                        ++row;
                        ++col;
                }
                last_d[k] = row;

                if (row == m && col == n) {
                        /* Hit southeast corner; have the answer. */
                        put_scr(script[k]);
                        exit(0);
                }
                if (row == m)
                        /* Hit last row; don't look to the left. */
                        lower = k+2;
                if (col == n)
                        /* Hit last column; don't look to the right. */
                        upper = k-2;
```

```
                }
                --lower;
                ++upper;
        }
        exceed(d);
}

/* in_file - read in a file and return a count of the lines */
in_file(filename, P)
char *filename, *P[];
{
        char buf[100], *malloc(), *fgets(), *save, *b;
        FILE *fp, *fopen();
        int lines = 0;

        if ((fp = fopen(filename, "r")) == NULL) {
                fprintf(stderr, "Cannot open file %s.\n", filename);
                exit(1);
        }
        while (fgets(buf, 100, fp) != NULL) {
                if (lines >= MAXLINES)
                        fatal("File is too large for fcomp.");
                if ((save = malloc(strlen(buf)+1)) == NULL)
                        fatal("Not enough room to save the files.");
                P[lines++] = save;
                for (b = buf; *save++ = *b++; ) /* copy the line */
                        ;
        }
        fclose(fp);
        return(lines);
}

/* put_scr - print the edit script */
put_scr(start)
struct edit *start;
{
        struct edit *ep, *behind, *ahead, *a, *b;
        int change;

        /* Reverse the pointers. */
        ahead = start;
        ep = NULL;
        while (ahead != NULL) {
                behind = ep;
                ep = ahead;
                ahead = ahead->link;
                ep->link = behind;      /* Flip the pointer. */
        }

        /* Print commands. */
        while (ep != NULL) {
                b = ep;
                if (ep->op == INSERT)
                        printf("Inserted after line %d:\n", ep->line1);
                else {  /* DELETE */
                        /* Look for a block of consecutive deleted lines. */
                        do {
                                a = b;
                                b = b->link;
                        } while (b!=NULL && b->op==DELETE && b->line1==a->line1+1);
                        /* Now b points to the command after the last deletion. */
```

```
                change = (b!=NULL && b->op==INSERT && b->line1==a->line1);
                if (change)
                        printf("Changed ");
                else
                        printf("Deleted ");
                if (a == ep)
                        printf("line %d:\n", ep->line1);
                else
                        printf("lines %d-%d:\n", ep->line1, a->line1);
                /* Print the deleted lines. */
                do {
                        printf("  %s", A[ep->line1-1]);
                        ep = ep->link;
                } while (ep != b);
                if (!change)
                        continue;
                printf("To:\n");
        }
        /* Print the inserted lines. */
        do {
                printf("  %s", B[ep->line2-1]);
                ep = ep->link;
        } while (ep != NULL && ep->op == INSERT && ep->line1 == b->line1);
    }
}

/* fatal - print error message and die */
fatal(msg)
char *msg;
{
    fprintf(stderr, "%s\n", msg);
    exit(1);
}

/* exceed - the difference exceeds d */
exceed(d)
int d;
{
    fprintf(stderr, "The files differ in at least %d lines.\n", d);
    exit(1);
}
```

REFERENCES

1. W. Tichy, 'The string-to-string correction problem with block moves', *ACM Trans. Comp. Systems*, **2,** (4), 309–321 (1984).
2. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Complete Problems*, W. H. Freeman, 1979.
3. M. J. Rochkind, 'The source code control system', *IEEE Trans. Software Engineering*, **1,** (4), 364–370 (1975).
4. W. Tichy, 'RCS — a system for version control', *Software — Practice and Experience*, **15**, (7), 637–654 (1985).
5. D. Sankoff and J. B. Kruskal, *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, 1983.
6. P. A. V. Hall and G. R. Dowling, 'Approximate string matching', *Computing Surveys*, **12,** (4), 381–402 (1980).
7. J. A. Gosling, 'A redisplay algorithm', *SIGPLAN Notices*, **16,** (6), 123–129 (1981).
8. N. Nakatsu, Y. Kambayashi and S. Yajima, 'A longest common subsequence algorithm suitable for similar text strings', *Acta Informatica*, **18,** 171–179 (1982).

9. R. A. Wagner and M. J. Fischer, 'The string-to-string correction problem', *Journal of ACM*, **21**, (1), 168–173 (1974).
10. A. V. Aho, D. S. Hirschberg and J. D. Ullman, 'Bounds on the complexity of the longest common subsequence problem', *Journal of ACM*, **23**, (1), 1–12 (1976).
11. E. W. Myers, 'An O(ND) difference algorithm and its variations', *Technical Report 85–6*, Department of Computer Science, University of Arizona, 1985.
12. D. S. Hirschberg, 'A linear space algorithm for computing maximal common subsequences', *Comm. ACM*, **18**, (6), 341–343 (1975).
13. J. W. Hunt and T. G. Szymanski, 'A fast algorithm for computing longest common subsequences', *Comm. ACM*, **20**, (5), 350–353 (1977).
14. J. W. Hunt and M. D. McIlroy, 'An algorithm for differential file comparison', *Computing Science Technical Report 41*, Bell Laboratories, 1975.